The Complexity of Problems

William Gasarch*
Univ. of Maryland

Abstract

We investigate problems that arise in computer science in terms of their execution times. We are concerned with both upper bounds (which are demonstrated by algorithms) and lower bounds (which are demonstrated by proof) on the execution time. The problems we study come from three realms. First, we investigate problems where getting an exact answer is important and can be done quickly. Finding the maximum element of a list is such a problem. Second, we investigate problems where getting a feasible solution (or showing that one cannot be obtained) is of interest. The problem of 2-coloring a graph is feasible, while the problem of 3-coloring a graph is thought to not be feasible. Third, we investigate problems that cannot be solved. Even in this realm one can have a notion of one problem being harder than another.

*Dept. of C.S. and Inst. for Adv. Comp. Stud., Univ. of MD., College Park, MD 20742, U.S.A. Supported in part by NSF grants CCR-8803641 and CCR-9020079 (email: gasarch@cs.umd.edu).

Contents

1 Introduction

A fundamental question in computer science is (informally)

"Given a problem A, how much time will it take to solve it?"

Typical problems are (1) sorting a list of numbers, (2) given constraints on availability of timeslots and locations, schedule classes to take exams, and (3) given a program, determine if it is going to halt.

We usually do not know exactly how much time it takes to solve a problem. Hence we try to get upper and lower bounds on the time. For x, an input for A, let n_x be an appropriate measure of the size of x. For example, if A is the problem of sorting a list of numbers then x would be a list and n_x would be the *number of elements* on the list. We can state our fundamental questions more formally as follows.

1. Find an algorithm $\mathcal{A}$ for A and a function U such that $\mathcal{A}(x)$ terminates *within* $U(n_x)$ steps. The function U is an *upper bound*.

2. Find a function L such that for *all* algorithms $\mathcal{A}$ for A there is some x such that $\mathcal{A}(x)$ will take *at least* $L(n_x)$ steps. The function L is a *lower bound*.

3. We would like L and U to be close to each other. If $L = U$ then we have pinned down the complexity of the problem exactly.

Item (1) is commonly done in computer science both formally and informally. Item (2) *requires* mathematical precision. For example, the statement "Sorting requires at least $n \log n$ steps" lacks precision since the notion of "step" is not well defined. A *model of computation* is needed. This model will specify what operations are allowed and how much they cost. In the case of sorting we will define a *decision tree* (see Definition 2.2) which counts a comparison as one step, data movement is free, and no other numeric operations are allowed. Having defined this model we can now make meaningful statements like "sorting requires at least $n \log n$ comparisons."

The model should be chosen such that all of the algorithms known for the problem fit the model. This leads to meaningful lower bounds that can be stated informally as follows:

"The types of algorithms that have been used to solve problem A in the past cannot take less than $L(n)$ steps."

This kind of statement is helpful in several ways.

1. No more energy should be wasted trying to obtain performance better than $L(n)$ using the usual approach.

2. If better performance is needed you may consider changing the problem.

3. The model and lower bound together clarify the assumptions and limitations of the underlying technology. This may lead to alternative technology. For example, if only comparisons are used then sorting n numbers *requires* $n \log n$ comparisons; however if arithmetic operations are allowed this can be improved.

In this paper we investigate problems in terms of models for algorithms for them, and upper and lower time bounds on those models. We will not present proofs, however references for such will be given. We study three realms.

1. **Decision Trees**. The decision tree model of complexity is appropriate when we want to know the *exact* complexity up to additive constants. A standard problem for this model is that of finding the maximum number from a list of numbers.

2. **P vs. NP**. P and NP are classes of problems to be defined later. They are appropriate for problems where our major concern is *feasibility*: can the problem be solved in reasonable time? Two examples: (1) Determining if a graph is connected can be solved in reasonable time. (2) Finding the fastest route through an entire network seems infeasible; most people do not think the problem can be solved in reasonable time.

3. **Recursive, recursively enumerable, and the arithmetic hierarchy**. These are classes of problems to be defined later. They are appropriate for problems where are major concern is *decidability*: can they be solved *at all*? In addition there are ways of taking undecidable problems and measuring just how undecidable they are.

1.1 Definitions and Notation

Notation 1.1 N denotes the set $\{0, 1, 2, \ldots\}$ of natural numbers.

Definition 1.2 Let f and g be functions from N to N.

1. $f = O(g)$ means that f is less than g if you ignore constants. Formally there exists c, n_0 such that

$$(\forall n \geq n_0)[f(n) \leq cg(n)].$$

2. $f = \Omega(g)$ means that f is greater than g if you ignore constants. Formally there exists c, n_0 such that

$$(\forall n \geq n_0)[f(n) \geq cg(n)].$$

3. $f = o(g)$ means that f is much smaller g. Formally $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Definition 1.3 If S is a set and $n \geq 1$ then S^n is a list of n elements from S. The elements need not be distinct.

2 Decision Trees

In this section we will study simple problems. The algorithms for these problem motivate a simple model of computation (decision trees) on which we can prove matching upper and lower bounds. These problems arise when compiling large data sets.

Here are the problems we will consider.

Definition 2.1 Let $1 \leq i \leq n$. Let S be an ordered domain (e.g., numbers using their natural order or words in alphabetical order). The following functions take n elements from S (formally an element of S^n) as input.

1. $\text{MAX}_n(x_1, \ldots, x_n)$ is the largest element of $(x_1, \ldots, x_n)$.
2. $\text{SEL}_n^i(x_1, \ldots, x_n)$ is the ith largest element of $(x_1, \ldots, x_n)$.
3. $\text{MED}_n(x_1, \ldots, x_n)$ is the $\left\lfloor \frac{n}{2} \right\rfloor$th element of $(x_1, \ldots, x_n)$.
4. $\text{HI-LO}_n(x_1, \ldots, x_n)$ returns both the maximum and minimum element of $(x_1, \ldots, x_n)$.
5. $\text{SORT}_n(x_1, \ldots, x_n)$ returns the elements of $(x_1, \ldots, x_n)$ in sorted order.

2.1 Finding the Maximum

The following algorithm finds $\text{MAX}_n(x_1, \ldots, x_n)$.

$tmax := x_1$
for $i := 2$ to n
  if $x_i > tmax$ then $tmax := x_i$
output$(tmax)$

The bulk of the algorithm's time is spent making comparisons. We will measure algorithms for MAX_n (and other problems) in terms of the *number of comparisons* made by the algorithm.

Definition 2.2 Let P be any problem whose input is $(x_1, \ldots, x_n) \in S^n$. A *decision tree* for P is a full binary tree labeled and interpreted as follows. (See Figure 1 for an example of a Decision tree that finds the maximum of 3 elements.)

1. Every non-leaf is labeled with a comparison of the form "$x_i < x_j$?"
2. Every leaf is labeled with a potential output.
3. Let $(x_1, \ldots, x_n) \in S^n$. We interpret the tree as an algorithm acting on $(x_1, \ldots, x_n)$ by thinking of YES as GO LEFT ON THE TREE, and NO as GO RIGHT ON TREE. This leads to a leaf of the tree that is labeled with the answer $P(x_1, \ldots, x_n)$.

Definition 2.3 A problem P *can be solved with* $f(n)$ *comparisons* if there exists a decision tree for P where no branch has more than $f(n)$ edges.

The algorithm for MAX_n can be phrased as a decision tree where each branch has $n-1$ edges. **Since we have a model of computation that captures the known algorithm we can pose the question of lower bounds intelligently**. This is given as:

Theorem 2.4 *[11]. Let* $n \in \mathbb{N}$.

1. MAX_n *can be solved with* $n-1$ *comparisons.*

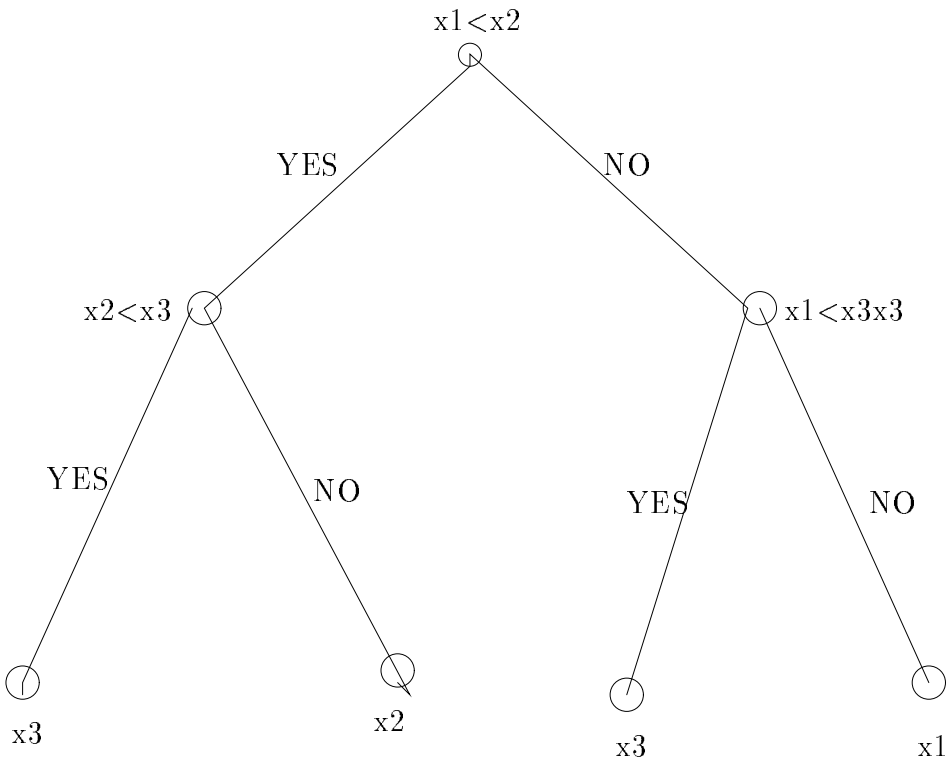

Figure 1

2. MAX_n *cannot be solved with fewer than* $n-1$ *comparisons. In fact, any decision tree that solves* MAX_n *has at least* $n-1$ *comparisons on every branch. Hence the best case and the average case require* $n-1$ *comparisons.*

To summarize, initially an algorithm for MAX_n was formulated that took $n-1$ comparisons. A model was developed to capture this algorithm, and in that model the algorithm is optimal.

2.2 Finding the ith largest element

The problem of MAX_n asked for the largest element of a list. We now look at the more general problem of finding the ith largest element of a list.

Consider the problem of finding the 2nd largest element of $(x_1, \ldots, x_n)$. The problem can be done in $2n-3$ comparisons: first find $x = \text{MAX}_n(x_1, \ldots, x_n)$ (using $n-1$ comparisons). Then find the maximum of $\{x_1, \ldots, x_n\} - \{x\}$ (using $n-2$ comparisons). Is this the best possible? NO! We present an algorithm that uses at most $n + \log n$ comparisons.

1. $\text{Input}(x_1, \ldots, x_n)$.

2. Form a tournament of pairs of comparisons: compare (x_1, x_2), (x_3, x_4), etc. Then compare the winners in pairs. Then compare the winners of that competition in pairs. Keep doing this until $y_1 = \text{MAX}_n(x_1, \ldots, x_n)$ is found. This will take $n-1$ comparisons. Note that this is not the algorithm we used in Section 2.1. While both algorithms find MAX_n in $n-1$ comparisons, this one leaves behind a residue of information that will be helpful in finding $\text{SEL}_n^i(x_1, \ldots, x_n)$.

3. Replace the element x_i that has the value y_1 in the tournament by $-\infty$. Redo the part of the tournament that this changes. This takes $\log n$ steps and yields the second largest element y_2.

Note that both algorithms given for finding the 2nd largest element can be expressed as a Decision tree. Hence the question of whether you can do better with 'the same kind of algorithm' can be expressed intelligently. The next theorem (the $i = 2$ case) states that the $n + \log n$ algorithm is essentially optimal.

Theorem 2.5 *[8, 17] Let i be a constant. Then the following happens.*

1. SEL_n^i *can be solved with* $n + (i-1)\log n + O(1)$ *comparisons. (This is similar to the algorithm we gave for finding the 2nd largest element.)*

2. *There exists a (small) constant c such that, for all i* SEL_n^i *cannot be solved with fewer than* $n + (i-1)\log n + c$ *comparisons.*

Both the algorithm and the lower bound can be modified to handle the following similar problem: given a list $(x_1, \ldots, x_n)$ find the 1st, 2nd, ..., and ith largest element. The upper and lower bounds are the same for this problem as they are for SEL_n^i except that the additive constant changes.

To summarize: we first obtained an algorithm for finding the 2nd largest element of a list that took $2n-3$ comparisons, and we then improved this to $n+\log n$. Both algorithms can be expressed as a decision tree. We then stated a theorem which gave matching upper and lower bounds for the general SEL_n^i problem, and implied that our algorithm for second largest is optimal.

2.3 MED_n

In the last section we looked at finding the ith largest element of a list. We thought of i as fixed and n as the measure of size. We now consider the problem of finding the $\left\lfloor \frac{n}{2} \right\rfloor$th element of a list (the middle or median element). This problem has a different character since i varies with n.

If the algorithm in Theorem 2.5 is used to find the median of $(x_1, \ldots, x_n)$ then this will take $n + \frac{n}{2}\log n = O(n \log n)$ comparisons. Can we do better? The lower bound in Theorem 2.5 does not apply here since i is not constant. We will exhibit an algorithm that finds MED_n in linear time.

Actually we present an algorithm that will, given $((x_1, \ldots, x_n), i)$, find the ith largest element of $(x_1, \ldots, x_n)$. We plan on having the program call itself recursively. The algorithm is from [2]:

1. Input$((x_1, \ldots, x_n), i)$. Let A be the array $(x_1, \ldots, x_n)$.

2. Group the elements into groups of 5. There are at most 4 elements left over. We will ignore them for purposes of exposition. (The choice of 5 is not important— any odd number larger than 3 would work.)

3. Find the median of every group of 5. (This can be done by the algorithm for SEL_n^i.) Put the set of medians into an array B of length roughly $\frac{n}{5}$. This step takes cn comparisons for some constant c.

4. Find the median x of the elements in B. This is done by calling this algorithm recursively.

5. Compare all elements to x. Put the elements that are less than x into an array $C_{<x}$, and all those that are greater than x into an array $C_{>x}$. Note that at this point we know j such that x is the jth largest element. If $j = i$ then output(x) and stop. (This could be done in a more space efficient manner.)

6. If $j < i$ then we know the ith largest element of A is in the ith largest element of $C_{>x}$. Call the algorithm recursively on $(C_{>x}, i)$.

7. If $j > i$ then we know the ith largest element of A is the $(j-i)$th element of $C_{<x}$. Call the algorithm recursively on $(C_{>x}, j-i)$.

One can show that $C_{<x}$ and $C_{>x}$ both have less than $\frac{7}{10}n$ elements. Hence, if T is the number of comparisons the algorithm makes on arrays of length n then

$$T(n) \leq cn + T(\frac{n}{5}) + T(\frac{7n}{10}).$$

$T(n)$ can be shown (by induction) to be $O(n)$.

This algorithm can be expressed by a decision tree. Hence the question of whether or not it is optimal can be raised intelligently. The following results provide upper and lower bounds on the problem; however, at this time no *matching* bounds are known.

Theorem 2.6 *Let $n \in \mathsf{N}$.*

1. MED_n *can be solved with* $5.43n + O(1)$ *comparisons (See [2]. This is essentially fine tuning the algorithm above.)*

2. MED_n *can be solved with* $2.97n + o(n)$ *comparisons. (See [7]. This is a rather complicated algorithm. The term* $o(n)$ *means a function of* n *that is substantially less than* n*.)*

3. *There is a constant* c *such that* MED_n *cannot be solved with less than* $2n - c\sqrt{n}$ *comparisons. [1]*

2.4 HI-LO$_n$

In Section 2.2 we briefly noted that we have upper and lower bounds for the problem of, given $(x_1, \ldots, x_n)$, find the 1st, 2nd, ..., ith largest elements. We consider a variant of this problem, the problem of finding the maximum and the minimum of a list.

The problem can be done in $2n-3$ comparisons: first find $x = \text{MAX}_n(x_1, \ldots, x_n)$ (using $n-1$ comparisons). Then find the minimum of $\{x_1, \ldots, x_n\} - \{x\}$ (using $n-2$ comparisons). Is this best possible? NO! The following algorithm solves the problem with approximately $\frac{3n}{2}$ comparisons [16]:

1. Input$(x_1, \ldots, x_n)$
2. Compare (x_1, x_2), (x_3, x_4), etc. This takes $\frac{n}{2}$ comparisons.
3. Let W be the set of elements that were greater and let L be the set of elements that were lesser in the comparisons from step 1. Note that W and L are both of size $\frac{n}{2}$.
4. Find the maximum element in W. This takes $\frac{n}{2} - 1$ and is also the maximum element in the original list.
5. Find the minimum element in L. This takes $\frac{n}{2} - 1$ and is also the minimum element in the original list.

Clearly this algorithm takes only $\frac{3n}{2} + O(1)$ comparisons. Since a "trick" allowed us to go from $2n-3$ comparisons to $\frac{3n}{2} + O(1)$ it is entirely plausible that another trick will lower the number of comparisons even lower. However this is not the case [16], as shown in the following:

Theorem 2.7 *[16] There exists a (small) constant c such that* HI-LO$_n$ *cannot be solved with fewer than $\frac{3n}{2} + c$ comparisons.*

To summarize: the "obvious" algorithm for finding HI-LO$_n$ element of a list turned out to *not* be optimal. Hence the need for lower bounds on the better algorithm is more acute than usual. Such a lower bound does exist. We end up with matching upper and lower bounds for the general HI-LO$_n$ problem.

2.5 SORT_n

The problem of sorting a list of numbers (or names) is one of the most important problems in computer science. As such it is also one of the most well studied. We will just make a few comments on it and urge the interested reader to look at references [11] and [5] for more detail.

Theorem 2.8 *1. The problem of sorting a list of n numbers can be solved in $O(n \log n)$ comparisons. (The constant is quite low.)*

2. There exists a constant c such that any algorithm for sorting requires at least $cn \log n$ comparisons.

2.6 Questioning the Model

Now that we have matching upper and lower bounds for several problems it is time to question the model. What if comparisons are replaced by some other type of query? Does this lead to faster algorithms? We can answer this with the following result [14]:

Theorem 2.9 *If $|S|$ is large compared to n then any lower bound obtained for decision trees will apply to decision trees that use any type of query of arity 2, i.e., a query that involves 2 elements at a time.*

The two assumptions to question are (1) what if queries of arity larger than 2 are used, and (2) what if $|S|$ is not large compared to n?

2.6.1 Allowing other types of queries

If your machine has powerful processors or perhaps parallelism then you may be able to do more interesting operations then comparisons in a unit time step. We consider now the possibility of being able to obtain the maximum of 3 elements as a basic query. With this model we can compute MAX_n substantially faster than $n-1$ operations.

1. Input$(x_1, \ldots, x_n)$.
2. Find the max of (x_1, x_2, x_3), (x_4, x_5, x_6), etc. This takes $\frac{n}{3}$ operations.

3. We now have $\frac{n}{3}$ candidates for the maximum. Repeat the procedure in step 2 to get it down to $\frac{n}{3^2}$. Keep doing this until only 1 element is left.

4. Output the element.

This algorithm's run time is bounded by $\frac{n}{3} + \frac{n}{3^2} + \cdots \leq \frac{n}{2}$. This is also a lower bound (up to an additive constant).

The above algorithm can be generalized to the case where each operation finds the max of k elements; the resulting algorithm runs in $\frac{n}{k-1}$ comparisons. This is also a lower bound (up to an additive constant). The above algorithm can be adjusted to obtain better upper bounds on all the other problems discussed in this section. For most of them the corresponding lower bound is unknown.

2.6.2 Cutting down the size of the domain

We look at the problem of finding the maximum in the case where the domain is small. We take an extreme case. Let the domain be $S = \{0, 1, 2\}$. The following algorithm clearly solves MAX_n: Go through the list checking, for each i, if $x_i = 2$. If every such an i is found then stop and output 2. If no such i is found then repeat the procedure checking if $x_i = 1$ ever occurs. If such is found then stop and output 1. If none are found then output 0.

The worst case of this algorithm is $2n$ checks. However the average case is quite good: less than 2 iterations. If S is small compared to n then similar algorithms to the one above can be worked out. There are inputs for which these algorithms take more than n comparisons, but they are far better in the average cases.

For the other problems discussed in this section there are also algorithms that work better than the lower bound if the domain is small. These also tend to have worse worst cases but far better average cases.

The problem of sorting can be solved faster than $O(n \log n)$ if we assume a bound on the input and use operations other than comparisons. We will not discuss this here but point the reader to the discussion of Counting sort and Radix sort in [5].

3 P vs. NP

In this section we will study problems that are more complicated then those in the previous section. The algorithms for these problems will be complicated and hence the model of computation is not easy to work with. In fact virtually no lower bounds are known. However there is a conjecture that seems to be true that would imply many lower bounds.

There are (literally) thousands of problems for which the model in this section is relevant. They come from many branches of computer science, applied mathematics, and mathematics. We will concentrate on two problems to motivate our model but will mention more problems later. Before describing the two problems we need a definition.

Definition 3.1 A *graph* is a set of *vertices* V and a set of *edges* which are unordered pairs of vertices. A *directed graph* uses ordered pairs. A *weighted graph* adds a cost to every edge in the graph. A *weighted directed graph* can easily be defined. The number of nodes in a graph is usually denoted by n and is our measure of the size of a graph.

Example 3.2 We give an example of a weighted directed graph. Let V be the set of cities in America. There is an edge (x, y) if there is an airline flying from x to y. The cost of the edge will be the cost of the flight.

We give two examples of problems on graphs.

1. The *s-t connectivity problem* (s-t) is the problem of, given a graph and two vertices s, t, determine if there is a way of getting from s to t in the graph.

2. The *Traveling Salesperson Problem* (TSP) is to determine the cheapest route through a weighted directed graph which visits every node exactly once. given a weighted graph, determine the minimal cost route that visits all vertices exactly once.

Although these problems look similar they will turn out to be very different. We examine algorithms for both of them.

Before presenting an algorithm for s-t we need the following definitions:

Definition 3.3 If two nodes are connected by an edge they are called *neighbors*. If x is a node then $nbhs(x)$ is the set of all the neighbors of x. If A is a set of nodes then $nbhs(A)$ is the set of all neighbors of vertices in A.

The following is an algorithm that solves the s-t problem:

1. Input(G, s, t) (G is a graph, s and t are nodes in the graph.)
2. Let $A_0 = \{s\}$ and $A_1 = A_0 \cup nbhs(A_0)$.
3. $i := 1$

   While $(A_i \neq A_{i-1})$ do

   $A_{i+1} := A_i \cup nbhs(A_i)$

   $i := i + 1$

   end
4. If $t \in A_i$ then output YES, else output NO.

Note that A_i will be the set of nodes that are of distance i from s. Hence the while loop is iterated less than $n + 1$ times. Each time it is iterated it takes at most n steps (this depends on how graphs are represented). Hence the algorithm runs in $O(n^2)$ steps. This is an informal statement since no model has been specified.

We now present an algorithm for TSP. First we need a definition.

Definition 3.4 If G is a graph then a *route* in G is a sequence of nodes $(v_1, \ldots, v_n)$ such that (1) every node of G is specified exactly once, (2) for all i, the nodes v_i and v_{i+1} form an edge, and (3) v_n and v_1 form an edge. If the graph is weighted then *cost of the route* is sum of the costs of the edges (v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n), and (v_n, v_1).

1. Input$(G = (V, E)$ (G is a weighted graph. We assume its nodes are $\{1, 2, \ldots, n\}$. We also assume that every pair of vertices forms an edge. If initially $\{v_1, v_2\} \notin E$ then we insert $\{v_1, v_2\}$ into the edge set and give it such a large weight that the minimum cost route will not use it. Note that every sequence of n distinct vertices is now a route.)

2. Keep two variables MINCOST and BESTROUTE. Initially BESTROUTE is the route $(1, 2, \ldots, n)$ and the MINCOST is the cost of that route.

3. Go through all possible routes. For each route R, if the cost of R is less than MINCOST then replace BESTROUTE with R and MINCOST with the cost of R.

4. Output BESTROUTE.

This algorithm looks at *all* possible routes. The number of routes is $n! = 1 \cdot 2 \cdot 3 \cdots n$. Hence the algorithm takes at least $n!$ steps. This is an underestimate because each iteration takes some time as well.

This algorithm is rather simplistic. It is a brute-force search over all possibilities. The execution time makes the algorithm infeasible. The questions arises as to whether there is an algorithm whose execution time is small enough so that the algorithm is feasible. Would an algorithm running in time $(n-8)!$ be feasible? Would an algorithm running in time $\sqrt{n!}$ be feasible? The following table shows that even though these functions are smaller than $n!$, they still grow to fast to make algorithms with those run times feasible. We assume that every step takes one nanosecond (10^{-9} seconds).

n	$\sqrt{n!}$	$(n-8)!$	$n!$
10	1.9×10^3 nanoseconds	2	3.6×10^6
20	$1.6 \times 10^9 = 1.6$ seconds	4.8×10^8	2.4×10^{18}
30	$1.6 \times 10^{16} = 12.4$ years	1.1×10^{21}	2.7×10^{32}
40	$9.0 \times 10^{23} = 6.9 \times 10^8$ years	2.6×10^{35}	8.2×10^{47}
50	$1.7 \times 10^{32} = 1.4 \times 10^{17}$ years	1.4×10^{51}	3.0×10^{64}
60	$9.1 \times 10^{40} = 7.0 \times 10^{25}$ years	8.0×10^{67}	8.3×10^{81}
70	$1.1 \times 10^{50} = 8.3 \times 10^{34}$ years	3.1×10^{85}	1.2×10^{100}
80	$2.7 \times 10^{59} = 2.0 \times 10^{44}$ years	6.1×10^{103}	7.6×10^{118}

A feasible algorithm would have to avoid the brute force search. In the next section we will define the notion of *polynomial time* which we believe pins down the notion of avoiding brute force search.

3.1 Polynomial Time and Nondeterministic Polynomial time

We need to define a model of computation that is flexible enough so that both algorithms given above can fit the model. Formally this is done with Turing machines or some other model of computation; (see [15, 18, 20]). We will omit the formal definition of a Turing machine but we will state several facts about them.

Fact 3.5 *1. A Turing machine consists of (1) a finite number of states to store information and (2) a tape to read input from, write output to, and use for intermediary storage. The tape is unbounded— we think of it as unlimited storage. There is a head that, at any point in a computation, is focused on one tape square. A program written for a Turing machine is essentially a set of instructions telling the head where to move (right, left, or stay where it is), what to write in that square, and what state to change to, given the current symbol it is scanning and state the machine is in.*

2. One can easily define what a step of a Turing machine is, hence one can define what it means for a computation to take a certain number of steps.

3. Any computer program can be expressed by a Turing machine.

4. There are several models of computation that tried to pin down the notion of "computable." All of them have a notion of what a step is and all of these models are equivalent to the Turing machine.

5. The equivalences of the models are efficient in the following sense: If M is one of these models then there exists a polynomial p such that if a problem A can be solved in time $t(n)$ in model M then A can be solved in time $p(t(n))$ on a Turing machine.

Since we will need a general definition we will talk about sets of strings. Strings can encode graphs, weighted graphs, and virtually anything else that might arise.

Definition 3.6 A *string* is a finite sequence of bits. If y is a string then $|y|$ is the length of y. The set of all strings is often denoted $\{0,1\}^*$.

Definition 3.7 A set A of strings is in P if there exists a Turing machine M and a polynomial p such that the following happens.

1. If $x \in A$ then $M(x)$ will terminate and output YES.
2. If $x \notin A$ then $M(x)$ will terminate and output NO.
3. If M is run on x then the computation terminates within $p(|x|)$ steps. By Fact 3.5.4 one can replace Turing machine by any of a number of models of computation. If a problem A is in polynomial time using a program (like the one for s-t) then A is in polynomial time using a Turing machine model, and hence A is in P. (P stands for Polynomial time.)

Polynomial time is a reasonable definitions of feasible. A brute force algorithm usually takes $n!$ or 2^n steps. Polynomial time indicates that we are *not* doing a brute force search. Hence some cleverness was involved. In many cases this cleverness can be used to fine tune the algorithm to your needs (e.g., the execution time can be reduced for the problems you care about, or for all problems). Hence polynomial time seems to be (empirically) a good definition of feasibility.

We would like to say the following: "We can now state the problem of whether or not there is a good algorithm for TSP formally: Is TSP in P?" However this is not quite true. We have defined P to be a set of *sets*, whereas the TSP problem is a *function*. Hence we redefine TSP as a set. From now on TSP will refer to the following problem: given (G, k) where G is a weighted graph and k is a number, does there exist a route for G that costs less than k. The following algorithm solves it:

1. Input(G, k).
2. Go through all possible routes. For each route R, if the cost of R is less than k then output YES and stop.
3. (If you reach this step then no route cost less than k.) Output NO.

This algorithm may look at *all* possible routes. The number of routes is $n! = 1 \cdot 2 \cdot 3 \cdots n$. Hence the algorithm takes at least $n!$ steps. This is an underestimate because each iteration takes some time as well. Note that

the time spend checking a particular route is small. In fact, that time is polynomial in n. Hence the reason this algorithm takes so long is that the number of elements in the search space is so large, even though evaluating any one of them is easy.

We can now phrase the lower bound question about TSP properly: Is TSP in P? This is an open problem in computer science; however, most computer scientists think that TSP is not in P. We look at the evidence for this. We will define a class called NP, but first we need to motivate it.

Imagine that someone gave you an instance (G, k) of TSP and claims that the instance is positive (i.e., there is a route taking less than k steps). Could he convince you of this? YES— he need only give you the route R. Given the route R it will not take much time for you to check that its cost is less than k. Note that this scenario is unrealistic— finding R is the hard part— but at least IF one had an R THEN one could check it. This leads to the definition of NP. The key intuition is that if x is a positive instance then there exists evidence for this which is easily verified.

Definition 3.8 A set A of strings is in NP if there exists a set B of ordered pairs of strings, and a polynomial p, such that

$$\begin{aligned} x \in A &\Rightarrow (\exists y)[|y| \leq p(|x|) \wedge (x, y) \in B]; \\ x \notin A &\Rightarrow (\forall y)[|y| \leq p(|x|) \Rightarrow (x, y) \notin B]. \end{aligned}$$

(The abbreviation NP stands for "nondeterministic polynomial time" which stems from an alternative definition. See [9].)

The string y serves as evidence that $x \in A$. Since $B \in P$ it is easy to determine if a particular string y is indeed evidence. Note that putting a problem in NP does not make it easy since there are still many (around $2^{p(|x|)}$) witnesses to check.

Example 3.9 TSP is in NP by using the set

$$B = \{((G, k), R) : R \text{ is a route in } G \text{ of cost less than } k \ \}.$$

Example 3.10 Let SAT be the set of all Boolean formulas for which there exists a satisfying truth assignment, that is, a way to assign TRUE and

FALSE to the variables in the formula such that the formula comes out TRUE. $SAT \in NP$ using the set B below:

$$B = \{(\phi, y) : \phi \text{ is a Boolean Formula, } y \text{ is a truth assignment}$$

$$\text{and } \phi(y) \text{ evaulates to TRUE}\}.$$

TSP and SAT are both in NP but we do not know which problem is harder, or even if either one is hard. We need a notion of being "the hardest problem in NP."

Definition 3.11 If A and B are both sets of strings then $A \leq_m^p B$ if there exists a function f that can be computed in polynomial time such that

$$x \in A \text{ iff } f(x) \in B.$$

Note that if $A \leq_m^p B$ and $B \in P$ then $A \in P$. (The "p" stands for polynomial time and the "m" stands for many-one, indicating that f can be many-one and need not be 1-1. This is not important for our discussion.)

We now define what if means to be NP-complete. NP-complete sets will be the hardest sets in NP in that if they are in P then everything in NP is also in P.

Definition 3.12 A set of strings A is NP-complete if the following hold.

1. $A \in NP$,
2. for all $B \in NP$, $B \leq_m^p A$.

Note that if A is NP-complete and $A \in P$ then $P = NP$.

This definition looks like it is very hard for a set to be NP-complete since *all* the sets in NP have to reduce to it. It also does not look very useful since we do not have any evidence (yet) that $P \neq NP$ so even if a set is NP-complete it could still be in P. In the next section we refute both these points.

3.2 The Relevance of NP-completeness

In 1971 Cook proved the following [4, 9] by using the formal definition of a Turing machine.

Theorem 3.13 *SAT is NP-complete.*

In 1972 Karp [10] proved 21 problems to be NP-complete. Since SAT was already known to be NP-complete Karp did not need to use Turing machines. To show that a set A was NP-complete he showed $A \in NP$ (this is usually trivial) and showed that, for some known NP-complete problem B (initially SAT), $B \leq_m^p A$. The problems Karp showed NP-complete were all natural problems. Since then thousands of problems have been shown to be NP-complete. Many of them are problems people have been trying to solve quickly for decades. If any of them are in P, then they are all in P. This is evidence that $P \neq NP$. Even though it is not mathematical evidence, it is still a highly plausible arguement: if $P = NP$ then one of those problems would probably have yielded to a polynomial time algorithm by now.

We now give some examples of problems that are NP-complete. The list is a sublist of the appendix of [9]. For references for where these problems were proven NP-complete, see [9]. Some of the entries on the list below are types of problems instead of actual problems.

1. GRAPH COLORING. Given a graph G, is it 3-colorable (that is, can you assign colors to the nodes of G so that you only use three colors and no two neighbors have the same color)? It is known that 2-coloring is in P.

2. CLIQUE. Given a graph G and a number k, does G have k points all pairs of which have an edge (such a set of points is called a clique)?

3. HAMILTONIAN PATH. Given a graph G, does there exist a path through the graph that hits every node exactly once?

4. NETWORK RELIABILITY. The input is a weighted graph G (weights are between 0 and 1, and we interpret them as probabilities of failure) and a number q (between 0 and 1). Assume the edges all fail with the assigned probabilities and that they are all independent of each other. The network *survives* if for every edge (x, y) there is a path from x to y

where none of the edges on the path fail. Determine if the probability of failure is less than q. (This problem is not known to be in NP, however it is known that if it is in P then $P = NP$.)

5. TRAVELING SALESPERSON PROBLEM. Discussed above. Many variations of it are also NP-complete.

6. INTEGER PROGRAMMING. Given an integer-valued matrix A and integer valued vector $\vec{b}$ does there exist an integer valued $\vec{x}$ such that $A\vec{x} \leq \vec{b}$. (The same problem with $\vec{x}$ allowed to be rational valued is in P. Both problems have many applications in business.)

7. DYNAMIC STORAGE ALLOCATION. The input is a set of triples (s, a, d) of natural numbers, and a natural number D. We think of each triple as an item to be stored which has size s, arrival time a, and departure time d. We think of D as a storage capacity. The problem is to determine if there is a way to allocate space for each item so that we never need more than D units of space. Two items can use the same space if they will be stored at nonoverlapping times. Once an item is put into a space it cannot be moved unless it is departing. (This problem models the problem of allocating space for processors as they come in. The problem here is actually easier since we know ahead of time the sizes, arrivals, and departures of the items.)

8. MINIMUM CARDINALITY KEY. The input is a set A of attribute names, a collection F of functional dependencies (ordered pairs of subsets of A), and a positive integer M. The problem is to determine if there a key of cardinality at most M for the relational system $\langle A, F\rangle$? (i.e., a minimal subset $K \subseteq A$ with $|K| \leq M$ such that the ordered pair (K, A) belongs to the closure F^* of F defined by (1) $F \subseteq F^*$, (2) $B \subseteq C \subseteq A$ implies $(C, B) \in F^*$,, (3) $(B, C), (C, D) \in F^*$ implies $(B, D) \in F^*$, and (4) $(B, C), (B, D) \in F^*$ implies $(B, C \cup D) \in F^*$.)

9. SCHEDULING: The input is a set of triples (t, r, d). We think of each triple as a task where t is how long the task takes, r is the release time (i.e., when the task will be ready to be worked on) and d is the deadline (i.e., upper bound on when you need to finish the task), The problem is to determine if there is a way to schedule the tasks to meet all the deadlines.

10. MIN AUT: Given two sets of strings S and T, and a number N, does there exist a finite automaton M with less than N states such that M accepts all the strings in S and rejects all the strings in T.

3.3 So your problem is NP-complete...

We have discussed several real problems that are NP-complete. Once you know that a problem you are working on is NP-complete what can you do?

1. Do not look for a polynomial time solution.
2. See if the problem you *really* want to solve may have some restrictions on it (e.g., only uses numbers that are less than 12). This new problem may have a polynomial time solution.
3. If there is a statement true about "most" of your inputs, then you may be able to use this to get a solution that is fast "most" of the time (e.g., most of the time the graph is planar, but not always).
4. If you do not need an exact solution then see if you can obtain an approximate solution.
5. Look in the literature for techniques specific to your problem (e.g., there has been much work done on the Traveling Salesperson Problem).

4 Decidable, Undecidable, and Beyond

Some problems cannot be solved at all. Even so, we will be able to intelligently compare and classify such problems. We will (or course) not use time or space as a measure of complexity. Instead we will use the number-of-quantifiers needed to describe the problem. For a more detailed treatment, and for definitions of terms we leave undefined, see [18, 20].

First we need to define what it means to be computable. Formally this is done with Turing machines or some other model of computation; however we will speak informally about programs. Fact 3.5 is strong evidence that every function that is computable is computable by a Turing machine. (This assumption is called Church's Thesis.)

Notation 4.1 Throughout this section $\varphi_0, \varphi_1, \varphi_2, \ldots$ will be a list of all (say) C++ programs. The notation $\varphi_i(x) \downarrow$ means that $\varphi_i(x)$ is defined; the notation $\varphi_i(x) \uparrow$ means that $\varphi_i(x)$ is undefined.

Definition 4.2 A *partial function* is a function which is allowed to be undefined on some points of its domain. If f is such a function then we denote that $f(x)$ is undefined by $f(x) \uparrow$, and that $f(x)$ is defined by $f(x) \downarrow$. A partial function is *total* if it is defined on all points of its domain.

Definition 4.3 A partial function f is *computable* if there exists a program φ such that (1) when $f(x)$ is defined $\varphi(x)$ halts and outputs $f(x)$, and (2) when $f(x)$ is undefined $\varphi(x)$ does not halt.

Definition 4.4 A set A is *solvable* if there exists e such that

$$\begin{aligned} x \in A &\Rightarrow \varphi_e(x) = 1 \\ x \notin A &\Rightarrow \varphi_i(x) = 0 \end{aligned}$$

Such sets are also called *decidable* or *recursive*. A set A is *unsolvable* if it is not solvable. Such sets are also called *undecidable* or *nonrecursive*.

4.1 Unsolvable problems

The following problems are unsolvable.

$HALT = \{(e,x) \mid \varphi_e(x) \downarrow\} = \{e \mid (\exists t)[\varphi_e(x) \text{ halts in less than } t \text{ steps }]\}$.

$TOT = \{e \mid \varphi_e \text{ is total}\} = \{e \mid (\forall x)(\exists t)[\varphi_e(x) \text{ halts in } t \text{ steps }]\}$.

$COF = \{e \mid \varphi_e \text{ halts on all but a finite number of values}\}$

Notice that there is a naive algorithm for $HALT$: given e run $\varphi_e(x)$ and, it if halts, output YES. The trouble is that if $\varphi_e(x)$ never halts this will not be discovered. The undecidability of $HALT$ says far more than that this particular algorithm will not solve $HALT$; it says that *no algorithm whatsoever* will solve $HALT$.

The examples above are essentially the problem of trying to tell if a program has a certain property. Generally such problems are unsolvable. We pin this down.

Definition 4.5 A set A is an *index set* if, for all x, y, if φ_x and φ_y compute the exact same partial function then either $x, y \in A$ or $x, y \notin A$. Hence the question of whether or not $x \in A$ depends only on the behavior of φ_x.

Theorem 4.6 *If A is an index set, $A \neq \emptyset$, and $A \neq \mathsf{N}$ then A is unsolvable.*

All the unsolvable problems encountered so far have to do with programs themselves. The next example is more natural, but it requires a brief background.

In 1900 David Hilbert, one of the leading mathematicians of his day, proposed 23 problems for future mathematicians to work on (see [3]). Even though the notions of decidability and undecidability were not known yet he stated (informally) that these problems should be solved or shown to be unsolvable. For the problems that asked for an algorithm this can be taken to mean that either an algorithm should be found or the problem should be proven unsolvable. For the problems that did not involve an algorithm a proof that it could not be solved this can be taken to mean some kind of independence result (i.e., neither the theorem, nor its negation, can be proven from the common axioms in use).

Hilbert's 10th problem (in modern terminology) was to devise an algorithm to determine, given a polynomial $p(x_1, \ldots, x_n)$ with integer coefficients, if there exists $a_1, \ldots, a_n \in \mathsf{N}$ such that $p(a_1, \ldots, a_n) = 0$. It is now known that no such algorithm exists [6, 12]. This can be phrased by saying that the set of polynomials with integer coefficients that have an integer solution is unsolvable. This is commonly stated as "Hilberts 10th problem is unsolvable." This is a *natural* example of an unsolvable problem since the concepts used to define it are not from computability theory (as opposed to $HALT$, INF and COF). There is a book on Hilbert's Tenth problem by Matijasevic [13] for non-logicians.

4.2 Logical Theories

We describe several problems whose decidability is of interest. These problems are not about programs hence they are more natural than $HALT$, INF, and COF.

Definition 4.7 A *logical language* contains the usual logical symbols ($\wedge$, $\vee$, $\neg$, $\exists$, and $\forall$) and variables that we think of as ranging over N. These variables are denoted by small letters (as opposed to capital letters). We will allow auxiliary symbols but they will be some subset of $\{+, \times, S, \leq\}$ where S is interpreted as successor ($S(x) = x + 1$). We may also allow second order quantifiers and variables that range over subsets of N. We denote second order variables by capital letters. We denote a language by the auxiliary symbols. If second order quantifiers are allowed then we will include "2" as an auxiliary symbol. For example $[\times, \leq]$ and $[S, \times, 2]$ both denote languages.

Definition 4.8 Let L be a logical language. A *sentence* in L is an expression using the symbols in L where every variable is quantified over. Note that every sentence is either TRUE or FALSE.

Example 4.9 1. Let $L = [\times, +, \leq]$. The following sentence means that there is no largest number. It is TRUE.

$$(\forall x)(\exists y)[x < y].$$

The following sentence asks if there is a solution to a certain polynomial. Since the quantifiers range over the natural numbers the sentence is asking for a solution in the natural numbers. It is FALSE since the left hand side is always greater than 0 (in fact, greater than 16).

$$(\exists x, y, z)[x^2 + 3xy + z^2 + 17 = 0].$$

2. Let $L = [S, \leq, 2]$. The following sentence says there exists an infinite set.

$$(\exists X)(\forall x)(\exists y)[x < y \wedge y \in X].$$

The following sentence says there exists two sets that partition the natural numbers.

$$(\exists X)(\exists Y)(\forall x)[(x \in X \vee x \in Y) \wedge (x \notin X \vee x \notin Y)].$$

A logical language is *decidable* if there is a program that will, given a sentence in that language, output YES if that sentence is TRUE, and NO if that sentence is FALSE. The following results are known.

1. The language $[S, \leq, 2]$ is decidable.
2. The language $[+, \leq]$ is decidable.
3. The language $[+, \leq, 2]$ is not decidable.
4. The language $[+, \times]$ is not decidable.

The undecidability of $[+, \times]$ can be derived from Gödel's incompleteness theorem (see any text in mathematical logic) as we briefly describe below.

Definition 4.10 An *axiom system* is a set of statements in a logical language. A *consistent axiom system* is a set of axioms such that it is impossible to derive a contradiction from it. A *decidable axiom system* is a set of axioms that forms a solvable set.

Gödel's Incompleteness Theorem is as follows.

Theorem 4.11 *If AX is a consistent decidable axiom system for $[+, \times]$ then there exists a statement S such that S is true of the natural numbers but is not provable from AX.*

From this we can derive that the language $[+, \times]$ is not recursively enumerable. We will later see that this language is far harder than that.

4.3 Recursively Enumerable Sets

Recall that we had a naive algorithm for $HALT$: given e run $\varphi_e(x)$ and if it halts then output YES. Modify this algorithm to also output the number of steps that were required. This algorithm still does not work since if $(e, x) \notin HALT$ then the algorithm does not halt. However, if $(e, x) \in HALT$, then this algorithm discovers *evidence* that $(e, x) \in HALT$, namely the number of steps needed for $\varphi_e(x)$ to halt. This is evidence in that the statement "$\varphi_e(x)$ halts in s steps" can be tested. The set $HALT$ has a nice property: if $(e, x) \in HALT$ then there is finite evidence for this fact. We generalize this property.

Definition 4.12 A set is *recursively enumerable* if it satisfies any of the following four equivalent conditions. We abbreviate recursively enumerable by r.e.

1. $A = \emptyset$ or A is the range of a recursive function. (This definition is why such sets are called "recursively enumerable"— we think of f as enumerating the set.)

2. There exists a solvable set $B \subseteq \mathsf{N} \times \mathsf{N}$ such that

$$A = \{x \mid (\exists y)[\langle x, y \rangle \in B]\}.$$

   We think of the y as being finite evidence that $x \in A$.

It is known (and not hard to prove) that if A is r.e. and not recursive then $\overline{A}$ is not r.e. This is used to prove certain sets are not r.e.

The following are examples of r.e. sets:

1. $HALT$

2. $\{e \mid \varphi_e \text{ halts on some prime }\} = \{e \mid (\exists x, t)[x \text{ is prime } \wedge \varphi_e(x) \text{ halts in } t \text{ steps }]\}$.

3. $\{e \mid \varphi_e \text{ halts on at least 100 numbers }\}$

The following are examples of sets that are not r.e. but their complements are r.e. Since we think of a set and its complement as being of the same complexity, we do not think of these are being harder than r.e. sets.

1. $\overline{HALT}$.

2. $\{e \mid \varphi_e \text{ halts on at most 100 numbers }\}$.

The following are examples of sets that are neither r.e., nor there complements are r.e. We think of these as being harder than r.e. sets. Let A be any of these sets. If $x \in A$ there may not be finite evidence for this, and if $x \notin A$ there may not be finite evidence for this either.

1. $\{e \mid \varphi_e \text{ halts on all primes }\}$.

2. TOT.

3. COF.

We look at the case of TOT more carefully. To show that $x \in TOT$ one would need to show that *for all* y $M_x(y)$ halts. There can be no finite evidence for this. To show that $x \notin TOT$ one would need to show that there is a y such that $M_x(y)$ does not halt. There can be no finite evidence for this either.

4.4 Reductions

We need a way to compare two problems, both of which may be undecidable, to each other. In particular we need a notation for the concept "If I had access to B then I could solve A."

Definition 4.13 $A \leq_T B$ if A can be solved given a "black box" for B. This is pronounced "A is *Turing-reducible* to B."

In the definition of $A \leq_T B$ we allow unlimited access to B. The following definition restricts that access by only allowing one query to B and saying how it is to be used.

Definition 4.14 $A \leq_m B$ if there exists a recursive function f such that $x \in A$ iff $f(x) \in B$. This is pronounced "A is *m-reducible* to B." The m indicates that the function f may be many-to-one. (This is an historical anachronism.)

4.5 The Arithmetic Hierarchy

We define a measure of difficulty of sets (that are already undecidable) based on the number of quantifiers required to define them. We will then state how complex the sets introduced in this section are in this measure.

Definition 4.15

1. $A \in \Sigma_1$ if there exists a solvable set $B \subseteq \mathsf{N}^2$ such that
$$A = \{x \mid (\exists y)[\langle x, y \rangle \in B]\}.$$
This is the same as r.e.

2. $A \in \Sigma_2$ if there exists a solvable set $B \subseteq \mathsf{N}^3$ such that
$$A = \{x \mid (\exists y)(\forall z)[\langle x, y, z \rangle \in B]\}.$$

3. $A \in \Sigma_n$ is defined similarly to Σ_1 and Σ_2.

4. $A \in \Pi_n$ if $\overline{A} \in \Sigma_n$.

5. A is in *the arithmetic hierarchy* if there exists n such that $A \in \Sigma_n$.

6. A is Σ_n-*complete* if $A \in \Sigma_n$ and $(\forall B \in \Sigma_n)[B \leq_m A]$. The notion of Π_n-*complete* is defined similarly.

The following are well known facts:

1. $\Sigma_1 \subset \Sigma_2 \subset \Sigma_3 \cdots$. (Proper inclusion.)
2. $\Pi_1 \subset \Pi_2 \subset \Pi_3 \cdots$.
3. $\Pi_0 = \Sigma_0$ and this is the class of all solvable sets.
4. $(\forall n \geq 1)[\Pi_n \neq \Sigma_n]$.
5. If $A \in \Sigma_1 \cap \Pi_1$ then A is recursive.
6. If $A \in \Sigma_2 \cap \Pi_2$ then $A \leq_T K$.
7. If $A \in \Sigma_n \cap \Pi_n$ then for all Σ_{n-1}-complete set B, $A \leq_T B$.
8. If A is Σ_n-complete and $B \leq_m A$ then $B \in \Sigma_n$. Hence $A \notin \Pi_n \cup \Sigma_{n-1}$.

We can now state the complexity of several sets introduced in this section.

1. $HALT$ is Σ_1-complete. Hence $HALT \in \Sigma_1 - \Pi_1$.
2. TOT is Π_2-complete. Hence $TOT \in \Pi_2 - \Sigma_2$.
3. COF is Σ_3-complete. Hence $COF \in \Sigma_3 - \Pi_3$.
4. Hilberts 10th problem can be expressed as a Σ_1-complete set. Hence it is in $\Sigma_1 - \Pi_1$.
5. Let TRUE be the set of true statements that can be expressed in the logical languages $[+, \times]$ or $[+, \leq, 2]$. The set TRUE is not in the arithmetic hierarchy.

5 Summary

The following tables summarizes most of the problems discussed in this paper together with what is known about there upper and lower bounds. After each table we interject commentary.

PROBLEM	UPPER BOUND	LOWER BOUND
Max	$n-1$	$n-1$
ith element.	$n+(i-1)\log n+O(1)$	$n+(i-1)\log n+c$
Median	$2.97n+o(n)$	$2n-c\sqrt{n}$
Hi-Lo	$\frac{3}{2}n$	$\frac{3}{2}n-c$
Sorting	$O(n\log n)$	$cn\log n$

For the above upper and lower bounds a decision tree model was used. For Max, ith largest, and Hi-Lo the best algorithm to use in the general case is the standard one presented in Sections 2.1, 2.2 and 2.4. The lower bounds indicate that these algorithms cannot be improved upon. If more information is known about the domain then a faster algorithm might be possible, as was shown in Section 2.6.2. The $2.97n$ median finding algorithm is not good in practice and one should actually use a fine tuned version of the algorithm presented in Section 2.3. Such an algorithm can be found in [2]. Sorting is similar to Max, ith largest, and Hi-Lo in that for the general case the standard algorithms (see [11] and [5]) are the best; however, for certain domains better algorithms exist (see [5]).

2-coloring graphs	P	linear
Graph connectivity	P	linear
s-t	P	linear
TSP	NP	NP-Complete
3-coloring graphs	NP	NP-Complete

The problems stated above that are in P have fast algorithms. The problems stated above that are NP-complete are not known to have fast algorithms. If the problem you are working on is NP-complete then see if you can get by with an approximation or with solving a less general problem. Your solution may be very domain-specific. For example, there are many techniques for solving the TSP problem, but they do not translate to being able to solve other NP-complete problems.

$HALT$	Σ_1	Σ_1-complete
TOT	Π_2	Π_2-complete
COF	Σ_3	Σ_3-complete
$TRUTH$	Σ_ω	Not in Arithmetic Hierarchy

The problems stated above are unsolvable. If you are faced with one of them then you may have to really scale it down quite a bit before you can solve it. For example, even though the halting problem is undecidable, if the language that the programs are written in is restricted then this sub problem may be solvable.

6 Acknowlegment

I would like to thank Adam Porter and Marv Zelkowitz for proofreading and commentary.

References

[1] Bent and John. Finding the median requires $2n$ comparisons. In *Proc. of the 17th ACM Sym. on Theory of Computing*, 1985.

[2] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7:448–461, 1973.

[3] F. E. Browder, editor. *Mathematical Developments Arising from Hilbert Problems*. American Mathematical Society, 1976. Two Volumes.

[4] S. C. Cook. The complexity of theorem proving procedures. In *Proc. of the 3th ACM Sym. on Theory of Computing*, pages 151–158, 1971.

[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 1990.

[6] M. Davis, H. Putnam, and J. Robinson. The decision problem for exponential diophantine equations. *Annals of Mathematics*, 74:425–436, 1961.

[7] D. Dor and U. Zwick. Selecting the median. In *Proc. 6th Annual Sym. on Discrete Algorithms*, 1995.

[8] Fussenegger and Gabow. A counting approach to lower bounds for selection problems. *J. ACM*, 26(2):227–238, April 1979.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, New York, 1979.

[10] R. Karp. Reducibilities among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computationsn*, pages 85–103. Plenum Press, 1972.

[11] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1973.

[12] Y. Matijasevic. Enumerable sets are diophantine (Russian). *Doklady Academy Nauk, SSSR*, 191:279–282, 1970. Translation in Soviet Math Doklady, Vol 11, 1970.

[13] Y. Matijasevic. *Hilbert's Tenth Problem.* MIT press, Cambridge, 1993.

[14] S. Moran, M. Snir, and U. Manber. Applications of Ramsey's theorem to decision tree complexity. *J. ACM*, 32:938–949, 1985.

[15] C. H. Papadimitriou and H. R. Lewis. *Elements of the Theory of Computation.* Prentice-Hall, Inc., 1981.

[16] Pohl. A sorting problem and its complexity. *Communications of the ACM*, 15, 1972.

[17] V. Pratt and F. F. Yao. On lower bounds for computing the i-th largest element. In *Proc. of the 14th IEEE Sym. on Found. of Comp. Sci.*, pages 70–81, 1973.

[18] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* McGraw Hill, New York, 1967.

[19] Schoenhage, Paterson, and Pippenger. Finding the median. *J. Comput. Syst. Sci.*, 13, 1976.

[20] R. I. Soare. *Recursively Enumerable Sets and Degrees.* Perspectives in Mathematical Logic. Springer-Verlag, Berlin, 1987.