SIENAcollege
Computer Science

Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2024

Topic Notes: Limitations of Algorithms

We conclude with a discussion of the limitations of the power of algorithms. That is, what kinds of problems cannot be solved by any algorithm, or which will require a minimum cost, and what is that minimum cost?

Lower Bounds

We will first look at *lower bounds*, which estimate the minimum amount of work needed to solve a given problem.

Once we have established a lower bound, we know that no algorithm can exist without performing work equivalent to at least that of the upper bound.

Some examples:

- the number of comparisons needed to find the largest element in a set of n numbers
- number of comparisons needed to sort an array of size n
- number of comparisons necessary for searching in a sorted array of n numbers
- the number of comparisons needed to determine if all elements of an array of n elements are unique
- number of multiplications needed to multiply two $n \times n$ matrices

Lower bounds may be exact counts or efficiency classes (big Ω). A lower bound is *tight* if there exists an algorithm with the same efficiency as the lower bound.

Some lower bound examples:

- sorting: lower bound $\Omega(n \log n)$, tight
- searching in a sorted array: lower bound $\Omega(\log n)$, tight
- determine element uniqueness: lower bound $\Omega(n \log n)$, tight
- n-digit integer multiplication: lower bound $\Omega(n)$, tightness unknown
- multiplication of $n \times n$ matrices: lower bound $\Omega(n^2)$, tightness unknown

There are a number of methods that can be used to establish lower bounds:

- trivial lower bounds
- information-theoretic arguments (decision trees)
- adversary arguments
- problem reduction

Trivial Lower Bounds

Trivial lower bounds are based on counting the number of items that **must** be processed in input and generated as output to solve a problem.

Some examples:

- Generating all permutations of a set of n elements has a trivial lower bound of $\Omega(n!)$ since all $n!$ permutations must be generated. This lower bound is tight since we have algorithms to do this that operate in $\Theta(n!)$.
- Evaluating a polynomial

  $$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$$

  requires that each of the n a_i's need to be processed, leading to a lower bound of $\Omega(n)$. Again, we have linear algorithms for this, so the bound is tight.
- Computing the product of two $n \times n$ matrices requires that each of the $2n^2$ numbers be multiplied at some point, leading to a lower bound of $\Omega(n^2)$. No known algorithm can meet this bound, and its tightness is unknown.
- A trivial lower bound for the traveling salesman problem can be obtained as $\Omega(n^2)$ based on the number of cities and inter-city distances, but this is not a useful result, as no algorithm comes anywhere near this lower bound.

One must take care in deciding how to count. One may think that to search for an element in a collection, the lower bound would involve looking at every element. That would lead us to a linear lower bound. But we know that in the case of a sorted array, we can use a binary search and find the element in logarithmic time. The key lies with the word "must" in the definition of a trivial lower bound. There is other information in that case (the ordering) that allows us to avoid ever considering many of the elements.

Information-Theoretic Arguments

Rather than the number of inputs or outputs to process, an *information-theoretic lower bound* is based on the amount of information an algorithm needs to produce to achieve its solution.

A binary search fits here – we are trying to find the location of a given value in a sorted array. Since we know the array is sorted, we can, with each guess, eliminate half of the possible locations of the goal, resulting in a lower bound (worst case) of $\log n$ steps.

Decision trees are a model of an algorithm's operation that can help us analyze algorithms such as search and sort that work by comparisons.

In a decision tree, internal nodes represent comparisons and leaves represent outcomes. The tree branches based on whether the comparison is true or false.

A simple tree for a search for the minimum among 3 numbers can be found in Figure 11.1 on p. 395 of Levitin.

- The number of leaves may exceed the number of outcomes if the same result can be obtained via different orders of comparisons.
- The number of leaves must be at least the total number of possible outcomes.
- The operation of an algorithm on a particular input is modeled by a path from the root to a leaf in the decision tree. The number of comparisons is equal to the number of edges along that path.
- Worst-case behavior is determined by the height of the algorithm's decision tree.

It quickly follows that any such tree with a total of l leaves (outcomes) must have $h \geq \lceil \log_2 l \rceil$.

Levitin Figures 11.2 (p. 396) and 11.3 (p. 397) show decision trees for selection and insertion sort of 3 elements.

Our main interest here is to determine a tight lower bound on comparison-based sorting:

- Any comparison-based sorting algorithm can be represented by a decision tree.
- The number of leaves (outcomes) must be $\geq n!$ to account for all possible permutations of inputs.
- The height of binary tree with $n!$ leaves $\geq \lceil \log_2 n! \rceil$.
- This tells us the number of comparisons in the worst case

$$C_{worst}(n) \geq \lceil \log_2 n! \rceil \approx n \log_2 n$$

  for **any** comparison-based sorting algorithm.
- Since we have an algorithm that operates in $\Theta(n \log_2 n)$ (merge sort), this bound is tight.

Adversary Arguments

Another approach to finding lower bounds is the *adversary argument*. This method depends on a "adversary" that makes the algorithm work the hardest by adjusting the input.

For example, when playing a guessing game to determine a number between 1 and n using yes/no questions (*e.g.*, "is the number less than x?"), the adversary puts the number in the larger of the two subsets generated by last question. (Yes, it cheats.)

The text also provides an adversary argument to show the lower bound on the number of comparisons needed to perform a merge of two sorted n-element lists into a single $2n$-element list (as in merge sort).

Problem Reduction

A key idea in the analysis of algorithms is *problem reduction*. If we can come up with a way to convert a problem we wish to solve to an instance of a different problem to which we already have a solution, this produces a solution to the original problem.

Suppose you wrote a program solving some problem A. A few days later, you find out a program needs to be written to solve a similar problem B. To avoid writing too much new code, you might try to come up with a way to solve B using your implementation of A.

So given your input to problem B, you would need to have a procedure to transform this input into corresponding input to an instance of problem A. Then solve the instance of problem A (which you already knew how to do). Then you need to transform the output of A back to the corresponding solution to B.

As a very simple example, suppose you have written a procedure to draw an ellipse.

```
draw_ellipse(double horiz, double vert, double x, double y)
```

This procedure likely deals with trigonometry and works at a low level with a graphics library. But it works, and that's all we know or care about.

If you are later asked to write a procedure to draw a circle. Hopefully you would quickly realize that you could make use of your solution to the problem of drawing an ellipse.

```
draw_circle(double radius, double x, double y) {
  draw_ellipse(2*radius, 2*radius, x, y);
}
```

So we have *transformed* or *reduced* the problem of drawing a circle to the problem of drawing an ellipse. We can say that `draw_circle` is "not more difficult than", or "can be transformed in polynomial time" to `draw_ellipse`.

For a somewhat more interesting example, suppose you are asked to solve the "pairing problem". You are given two n-element arrays $A1$ and $A2$. Your task is to rearrange the values in $A2$ such that the smallest value in $A2$ is paired with the smallest value in $A1$. The second smallest in $A2$ is paired with the second smallest in $A1$, and so one. Only values of $A2$ are rearranged; $A1$ is unchanged.

So for the input arrays

```
A1  23   5  57  45
A2 150 175 100 120
```

the output would be

```
A1  23   5  57  45
A2 120 100 175 150
```

We aren't concerned about the details of a solution, let's just assume we have a solution to this problem. But now, you are asked to solve a different problem. It's one we know well: sorting an array A containing n values.

How can we make use of the solution to the pairing problem to solve the sorting problem?

We can transform the sorting problem into an instance of the pairing problem by using the input array A from the sorting problem as array $A2$ in the pairing problem, and creating an already-sorted array (probably just containing the values $1, 2, \cdots, n$) and using that as $A1$. Application of the pairing problem's solution will result in the sorting of $A2$, which is exactly what we wanted.

What is the total cost? It's $O(n+T(n)+1)$ - where the first n is the time it takes to transform from the sorting problem to the pairing problem (the construction of $A1$), $T(n)$ is the cost of computing the solution to the pairing problem, and 1 (a constant) is the cost of transforming back to a solution of the sorting problem (which in this case is trivial).

So we have *reduced* the sorting problem to an instance of the pairing problem.

Such a problem reduction can be used to show a lower bound.

- If problem A is **at least as hard as** problem B, then a lower bound for B is also a lower bound for A.
- Hence, we wish to find a problem B with a known lower bound that can be reduced to the problem A.

In our example, problem A is the pairing problem and problem B is the sorting problem. The sorting problem has a known lower bound of $\Omega(n \log n)$. Since the sorting problem can be reduced to an instance of the pairing problem, the pairing problem is at least as hard as the sorting problem, meaning the pairing problem also has a lower bound of $\Omega(n \log n)$.

Think about this for a minute and it should make sense: if we know that **any** solution to a problem has to have some minimum cost (the lower bound) and we show that some other problem is at least as hard as that problem, that other problem shares the lower bound of the first.

Important reminder: just because we show that a problem is in some $\Omega(g(n))$, this does not mean it is also in $\Theta(g(n))$.

As a more interesting example, suppose we wish to find a lower bound for the problem of finding the minimum spanning tree of a set of points in the plane. This problem, known as the Euclidean MST problem, is defined as follows: given n points in the plane, construct a tree of minimum total length whose vertices are the given points.

The problem with the known lower bound we'll use is the element uniqueness problem ($\Omega(n \log n)$, tight).

So our task is to reduce the element uniqueness problem to an instance of the Euclidean MST problem. We proceed as follows:

- If our input to the element uniqueness problem is a set of numbers $x_1, x_2, \cdots, x_n$, we can transform these to a set of points in the plane by attaching a y-coordinate of 0 to each: $(x_1, 0), (x_2, 0), \cdots, (x_n, 0)$.
- If we then solve the Euclidean MST problem on this set of input to obtain a spanning tree T.
- From this, we can obtain a solution to the original element uniqueness problem by checking for a 0-length edge.

So we can deduce a lower bound of $\Omega(n \log n)$ for the Euclidean MST problem.

Tractable Problems, P and NP

A problem is said to be *tractable* if there exists a polynomial-time ($O(p(n))$ where $p(n)$ is a polynomial of the input size n) algorithm to solve it.

A problem for which no such algorithm exists is called *intractable*.

When attempting to determine the tractability of a problem, the answer may be:

- Yes, it is tractable. This is shown by producing a polynomial-time algorithm.
- No, it is not tractable. This is done by proof that no algorithm exists or that any algorithm must take exponential time. Example: Towers of Hanoi. We have to make all 2^n moves.
- The answer is unknown.

Before we continue, we make a distinction between two problem types: optimization problems and decision problems.

In an optimization problem, we look to find a solution that maximizes or minimizes some objective function. For a decision problem, we seek the answer to a yes/no question.

Many problems have both decision and optimization versions. For example, the traveling salesman problem can be stated either way:

- optimization: find a Hamiltonian cycle of minimum length.
- decision: find Hamiltonian cycle of length $\leq m$.

Decision problems are more convenient for formal investigation of their complexity and our discussion that follows will assume decision problems.

P and NP

We define class P as the class of decision problems that are solvable in $O(p(n))$ time, where $p(n)$ is a polynomial of problem's input size n.

Many of the problems we have seen fall into class P, but do all decision problems fall into this class?

The answer is no. Some problems are *undecidable*, such as the famous *halting problem*. The problem: given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

We can prove by contradicion that this problem is undecidable.

Suppose that A is an algorithm that solves the halting problem. More formally, for any program P and input I, $A(P, I)$ produces a 1 if P halts when executed with input I and 0 if it does not.

Now, take a program P and use the program as its own input. We'll use the algorithm A to construct another program Q such that $Q(P)$ halts if $A(P, P) = 0$ (P does not halt on input P) but does not halt (goes into a loop) if $A(P, P) = 1$ (P halts on input P).

And finally, we apply Q to itself: $Q(Q)$ halts if $A(Q, Q) = 0$ (program Q does not halt on Q) and does not halt if $A(Q, Q) = 1$ (program Q halts on Q).

Given our construction of the program Q, neither of these outcomes is possible, so no such algorithm A can exist.

There is also a set of problems for which it has been shown to take exponential time to obtain a solution (with a provable lower bound).

But a larger and important set of problems have no *known* polynomial-time solution, but there is no proof that no such solution exists.

We have seen some of these problems:

- Hamiltonian circuit
- Traveling salesman

- Knapsack problem
- Partition problem
- Bin packing
- Graph coloring

For some of these, while there is no known polynomial-time solution, we can easily check if a given candidate solution is valid. This leads us to..

Class NP (nondeterministic polynomial) is the class of decision problems whose proposed solutions can be verified in polynomial time, *i.e.*, are solvable by a *nondeterministic polynomial algorithm*.

A nondeterministic polynomial algorithm is an abstract procedure that:

1. generates a random string purported to solve the problem
2. checks whether this solution is correct in polynomial time

By definition, it solves the problem if it is capable of generating and verifying a solution on one of its tries.

Many decision problems are in NP, including all of those that are in P.

The big open question in theoretical computer science is whether $P = NP$. What would it mean?

First, one more definition.

A decision problem D is *NP-complete* if it's as hard as any problem in NP, *i.e.*,

- D is in NP, and
- every problem in NP is polynomial-time reducible to D

The first requirement isn't bad – just produce a nondeterministic polynomial algorithm. The second, known as the *NP-Hard* property, is pretty daunting. We're supposed to show that **every** problem in NP is polynomial-time reducible to this problem?

Of course, any NP-complete problems are polynomially reducible to each other, so it suffices to show that we can reduce any one problem in the set of NP-complete problems to a problem to show it is NP-complete.

Nonetheless, there are problems known to be NP-complete.

Informally, an NP-complete problem is one for which we have not yet found any $O(n^c)$ algorithms, **and** if we do find an $O(n^c)$ algorithm to solve it, we'll then get $O(n^c)$ solutions to **all** problems in NP.

Figure 11.6 on p. 406 of Levitin shows the idea graphically.

The first problem shown to be NP-complete was the *CNF-satisfiability problem*: Is a boolean expression in its conjunctive normal form (CNF) satisfiable, *i.e.*, are there values of its variables that makes it true?

For our purposes, we will just note that this problem is in NP by noting this nondeterministic algorithm:

1. Guess truth assignment
2. Substitute the values into the CNF formula to see if it evaluates to true

A check can be done in linear time.

The deterministic solution requires 2^n evaluations.

For example, consider the expression:

$$(A|\neg B|\neg C)\&(A|B)\&(\neg B|\neg D|E)\&(\neg D|\neg E)$$

We would have to check each of the $2^5 = 32$ combinations of boolean values of A, B, C, D, and E.

Other problems can be shown to be NP-complete by producing a reduction of CNF-Sat to that problem. That is, if a problem can be used to solve CNF-Sat, the problem is NP-complete.

Some Famous NP-Complete Problems

- The *Independent Set Problem*.

  Input: An undirected graph G and a value k.

  Output: Yes if G has an independent set of size at least k. An independent set is a subset of the vertices such that no pair of vertices has an edge between them.

  An algorithm to solve this: Enumerate every possible subset and check if it forms an independent set. Keep track of the largest such subset.

  In the worst case, $2^{|V|}$ subsets are searched.

  This is the best known solution, but even for a problem with 60 vertices and a computer that can do 1 billion subsets per second, it would take 32 years to solve the problem!

  What happens if we move up to 61 vertices?

- The *Hamiltonian Cycle Problem*.

  Input: An undirected, weighted graph $G = (V, E)$.

  Output: Yes if G has a Hamiltonian cycle (a cycle that visits every vertex exactly once), no otherwise.

An algorithm to solve this: Enumerate every possible cycle of vertices and check if the edges that connect it exist.

In the worst case, we need to check $|V|!$ paths.

This is the best known solution, but again for a relatively small problem – 20 vertices, and a computer that could do 1 billion permutations per second, again we're looking at 32 years!

What happens if we move up to 21 vertices?

So, Does $P = NP$

Sure, if $P = 0$ or $N = 1$. But that's not helpful.

Most theoretical computer scientists believe that no polynomial time solutions exist for the class of NP-complete problems.

There are hundreds of NP-complete problems known, and in the nearly 50 years since the problem was posed in 1971, no one has found a polynomial time solution to any of them, nor has anyone proven than no such algorithms can exist.

Yet, it remains a central open question in computing.

Dealing with NP-Hard Problems

Realistically, NP-hard problems are "solved" by approximations or stochastic approaches. See Chapter 12!