CS-786 Randomized Algorithms 2 August 2024

Lecture 01: Limitations of Computation

Lecturer: Rohit Gurjar *Scribe: Prerak, Ayush*

1 Lower Bound on Sorting

Theorem 1.1. *Given an array of n distinct numbers, any comparison based algorithm (an algorithm that can only make comparison between two elements to obtain an ordering amongst them, and does not use the value of the elements in any way) needs to perform at least $\theta(\log_2(n!)) = \theta(n \log_2(n))$ comparisons.*

Proof of Theorem 1.1. Before we begin with the proof, let us first establish an equivalence between sorting and obtaining the permutation of the input array.

Claim 1.2. *Note: When we say permutation of an array, we mean the permutation which transforms the sorted version array to the given array.*

Given a comparison based algorithm that determines the permutation of an input array that performs at most $f(n)$ comparisons, we can construct a sorting algorithm which performs $\theta(f(n))$. In the reverse direction, given a comparison based algorithm that sorts an input array using at most $f(n)$ comparisons, we can construct an algorithm that determines the permutation of an input array.

Proof of Claim 1.2.

In the forward direction, given a sub-routine that determines the permutation of an input array after performing $f(n)$ comparisons, one can first call this subroutine which returns the permutation, and using this permutation, one can return the sorted array without any more queries to the array. Hence, we can sort the array in $\theta(f(n))$ comparisons.

In the backward direction, consider augmenting the elements with their positions as well, that is we construct a new array where the i^{th} element is (a_i, i). Sorting this new array using the sorting algorithm, using the comparison of two elements in this new array as comparing the first number in the two tuples, we can obtain the permutation from the sorted array by then removing the first number in each tuple, and computing the inverse of the resulting permutation (which does not require any further queries to the array). Hence, we can compute the permutation using $\theta(f(n))$ queries if the sorting algorithm performed $f(n)$ queries.

■

With the above claim now, we have established that up to a constant, the two problems are equivalent in terms of minimum comparisons required, and hence it suffices to provide a lower bound for obtaining the permutation of the input array. To that extent, consider any algorithm that finds the permutation of the input array. Since this algorithm has to work for all inputs, it suffices to provide one adversarial input on which the algorithm requires $\Omega(n \log_2(n))$ comparisons.

Let S_0 be the set of all possible inputs (permutations) of n sized array, with $|S| = n!$. At each comparison step, the algorithm queries by providing indices i, j. Let us define set S_k to be the set of all inputs that are consistent with the results of the queries in steps $1, 2, \dots, k$. The algorithm can output an answer at step k_f only when $|S_{k_f}| = 1$. For a set S_k and query q, define a partition of $S_k = S_k[q_t] \cup S_k[q_f]$ where $S_k[q_t]$ represents the inputs from S_k where the query holds true, and $S_k[q_f]$ as the inputs where query holds false. It is easy to see that:

$$S_{k+1} = \begin{cases} S_k[q_t] & \text{if the result of query was true} \\ S_k[q_f] & \text{if the result of query was false} \end{cases} \tag{1.1}$$

Consider an adversary that, on query q at $(k+1)^{th}$ step, returns true if $|S_k[q_t]| \geq |S_k[q_f]|$ and false otherwise. It follows from 1.1 that $|S_{k+1}| \geq 1/2|S_k| \implies |S_k| \geq n!/(2^k)$. Hence, if $|S_{k_f}| = 1$, we have $k_f \geq \log_2(n!)$.

This shows that for any behavior of the algorithm, we can generate an adversarial input on which the algorithm needs atleast $\log_2(n!)$ comparisons to obtain the permutation.

■

The above bound is an example of an information theoretic lower bound: We showed that we need at least $f(n)$ queries to obtain enough information about the input. And no matter how much computation power one can use, without $\log_2(n!)$ comparisons, one will not get enough information to sort the array (assuming that the algorithm gets no information about elements other than the ordering among the queried pair).

Problem 1.3 (Homework)**.** *Consider the following problem: You are given n numbers, and exactly two of them are equal. You are allowed to compare two numbers $A_i \square A_j$, with the result being: $<, >$ or $=$. The task is to find the the pair that is equal.*

Show that any algorithm requires $\Omega(n \log_2(n))$ comparisons, and give an algorithm that does it in $O(n \log_2(n))$ comparisons.

However, the kind of bounds that we shall be more interested in are complexity lower bounds: Given all the information that we need about the input, how much computation is required to obtain the answer.

But before we answer this, a more important question to think about is: Can we even "compute" an answer for any given problem? As an example, consider the problem of weather forecasting. Let us look at some of the limitations we might face when trying to solve the problem:

- Lack of understanding: Maybe we are not experts in meteorology, and we do not understand the exact science behind how weather forecasting works.
- Lack of information: Maybe we do understand how weather works, but we do not have access to all the variables required to accurately predict the weather.

- Lack of computation power: Even with the above two, we might just not have enough computational power to solve the problem.

In this course, we shall assume that the first two are not limitations, that is we completely understand the objects and have complete information about the input. However, even if we overcome the first two limitations, and assume infinite computational power, is it still possible to solve all the problems?

To answer this question, let us first formalize what a "problem" is:

2 Computational Task

Definition 2.1. *A computational task consists of an input $x \in \{0,1\}^*$, and a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ (or a function $f : \{0,1\}^* \to \mathcal{P}(\{0,1\}^*)/\phi$ if one wants all the inputs to have some output). The task is to return an output y such that $(x, y) \in R$.*

We are interested in two broad categories of computational tasks:

1. **Search Problem**: A task where the output $y \in \{0,1\}^*$
2. **Decision Problem**: A task where the output $y \in \{0,1\}$

It turns out that for all the search problems that we would be looking at, we can define a "natural" or "equivalent" decision problem (in loose terms) such using one as a sub-routine, one can solve the other with an at-most polynomial overhead (that is, the problem can be solved by performing polynomial steps, where each step is a computational step or a call to the sub-routine).

Example 2.2. *SAT*

Input: A propositional formula ϕ. Output: A satisfying assignment σ for ϕ, and some default value if ϕ is not satisfiable.

If we try to define our decision problem as: Given a formula ϕ and an assignment σ does σ satisfy ϕ, we do not know of a way to solve the search problem from the decision problem with at most a polynomial overhead, that is, this is not an example of a natural decision problem associated to the search problem.

Here is an example of a natural decision problem for SAT: Given a formula ϕ, is it satisfiable? With this sub-routine, one algorithm to solve SAT is as follows:

- *If ϕ is not satisfiable, return the default value.*
- *Else, we shall maintain a partial assignment for the propositional variables, and keep adding a new assignment to it after each step. We iterate over variables, and at each step i, we assign x_i an arbitrary value (say 0). Consider the formula obtained after the substitution*

for assigned variables, and ask the sub-routine if the new formula is satisfiable. If so, we do not modify the assignment for x_i, and otherwise we assign x_i as the negation of its initial assignment (1 in this case).

It is easy to see that the above algorithm correctly returns a satisfiable assignment if ϕ is satisfiable, and runs in time polynomial to the size of the formula.

However, to show that the above defined decision problem is in fact "equivalent" to the search problem, we also need to show the reverse direction: Using the search problem, we can solve the decision problem with an atmost polynomial overhead. But this is easy, since this can be done by checking if the returned assignment is the default value which corresponds to an unsatisfiable formula.

Example 2.3. *PRIME FACTORS*

Input: A non-negative integer n (Note that the input size is $\log(n)$) Output: Prime factors of n

If we try to define the decision problem as: Given a prime p and an integer n, does p divide n, there again does not appear to be an efficient algorithm which can solve the search problem using the decision problem. If we try to iterate over all the primes from 1 to $n-1$, we would need $\theta(n/\log(n))$ (approximately the number of primes from 1 to n) queries to the sub routine, which is not polynomial in input size.

Instead consider the following decision problem: Given a number n and a number k, is there a prime factor of n less than k (and greater than 1)?

Using this subroutine, we can perform a binary search to obtain the smallest prime factor of n, keep dividing n by p until we obtain a number n' not divisible by p, and recursively find the prime factors of n'. Since the number of prime factors of n are at most $\log_2(n)$ ($n = \prod_i p_i^{\alpha_i} \geq \prod_i 2 = 2^{\# \ primes}$), and in each recursive call, we perform poly($\log(n)$) steps, the overall algorithm runs in time polynomial to the size of the input.

The other way around, given an algorithm for the search problem, to solve the decision problem, we just iterate over all the prime factors of n, and if we find a factor less than k, we return YES, else we return NO.

We still need to define what it means to perform computation, without a proper definition, every problem can trivially be computed, by assuming our model of computation as a "black-box" which magically outputs the correct answer. However, we defer the formal definition for this to a later lecture. Here we shall assume that the computations are performed by `C++` programs.

With all the definitions, let us now try to answer the question: Given a problem, can we always write a program to solve it?

3 Undecidable Problems

Claim 3.1. *You cannot write a program for solving every problem*

We will give two different proofs for the same. One will be based on the counting argument while other will be based on technique called diagonalization.

Proof of Claim 3.1 using counting. Firstly, we will give the proof based on counting. As two different decision problems can't have the same program, it's sufficient to show that the cardinality of the set of decision problems is greater than that of the set of programs.

Each C++ program can be represented as a binary string in some suitable encoding (like ASCII). Since each binary string can be interpreted as a natural number (in binary representation), this establishes a injection between the set of all C++ programs and the set of natural numbers ($\mathbb{N}$). Injection of an infinite set with $\mathbb{N}$ implies bijection, therefore we have established bijection between the set of all C++ programs and the set of natural numbers

Now, consider problems as decision problems (yes/no questions) over the set of all possible inputs (encoded as binary strings). The set of all possible problems can be viewed as the set of all functions mapping binary strings to $\{0, 1\}$. Each such function can be represented by an infinite binary sequence, where the i-th bit of the sequence corresponds to the output of the function on the i-th binary string.

The set of infinite binary sequences has the same cardinality as the set of real numbers ($\mathbb{R}$), because each real number in the interval $(0, 1)$ can be uniquely represented by an infinite binary sequence (its binary expansion).

To extend this bijection to all real numbers, we can use the function

$$f(x) = 2x - 1$$

which maps $(0, 1)$ bijectively onto $(-1, 1)$, and then extend it to $\mathbb{R}$ by applying another transformation, e.g.,

$$g(x) = \frac{x}{1 - |x|}$$

which maps $(-1, 1)$ onto $\mathbb{R}$. This shows that there is a bijection between the set of infinite binary sequences and the set of real numbers $\mathbb{R}$. Therefore, there is a bijection between the set of problems and the set of real numbers.

Since the set of natural numbers ($\mathbb{N}$) is countable and the set of real numbers ($\mathbb{R}$) is uncountable, the cardinality of the set of real numbers is strictly greater than that of the set of natural numbers. Therefore, the set of problems is larger than the set of programs, proving that there exist problems for which no program can be written.

This completes the proof using the counting argument. ■

The above proof only shows that there exists problems for which we cannot write a program. But are there such explicit problems? The below proof shows how one can find such explicit problems. Recall that above we have shown a bijection between set of natural numbers and set of all C++ programs. Thus, we can talk about an enumeration of the programs as in: 1st program, 2nd program, $\ldots$, ith program, $\ldots$.

Proof of Claim 3.1 using Diagonalization. Consider a function $g : \mathbb{N} \to \{0,1\}$ defined as follows:

$$g(i) = \begin{cases} 1 & \text{if the } i\text{-th program halts on input } i, \\ 0 & \text{otherwise.} \end{cases}$$

We claim that there is no program that computes the function g.

To show this we will take an arbitrary program G and argue that the output of program G must differ from function g on some input.

Let G be an arbitrary program. Now, consider the following program P:

Program P:

- Input: $i \in \mathbb{N}$
- Run program G on input i.
- If $G(i) = 1$, then enter an infinite loop.
- Otherwise, return 0.

Let j be the index of program P in the enumeration of all programs. Now, consider the value of function $g(j)$ and output of program G on input j.

- If $g(j) = 1$: According to the definition of g, this means that program P halts on input j. But program P can halt only when the output of G on input j is different from 1. Hence, $G(j) \neq 1$.
- If $g(j) = 0$: This implies that program P does not halt on input j. That can happen in two ways: (i) program G does not halt on input j or (ii) program G outputs 1 on input j. In either case, we have $G(j) \neq 0$.

Here, we have shown that no program agrees with function g on all inputs. In other word, there is no program to compute function g.

This completes the proof using diagonalization. ■

Using the above claim, now we will prove that there is no program for the halting problem. In other words, the halting problem is undecidable.

Halting Problem:

Given a description of a program P and an input x, determine whether P halts (i.e., finishes execution) when run with input x, or P runs forever.

Formally, define a function h such that:

$$h(P,x) = \begin{cases} 1 & \text{if program } P \text{ halts when run with input } x, \\ 0 & \text{if program } P \text{ does not halt when run with input } x. \end{cases}$$

Proof of Undecidability: If there is a program for computing function h, then we can easily get a program for computing function g defined above. For input i for g, we simply need to find a description P of the ith program and give input (P, i) to the program computing function h. As we have shown that there is no program to compute function g, the same is true for function h.

HW

Problem: Prove that the following problem is undecidable: Given two C++ programs, determine whether they have the same behavior (i.e., whether they produce the same output for all possible inputs).