MIT 6.006 Introduction to Algorithms

Lecture 1
Algorithms and Computation

David Lin

December 2021

1 Introduction

What is this course about?

1. Solve computational problems
2. Prove correctness of an algorithm
3. Argue efficiency
4. Communication of these ideas

2 Computational Problem

When we say computational problem, what does it mean?
A computational problem might be something that we want to compute. But specifically, a computational problem is having a set of input and output, and the problem is a binary relation to these inputs and output. So for each input, I can specify which output is correct. We usually have some predicates so that if I have an input, I can check if an output is correct.

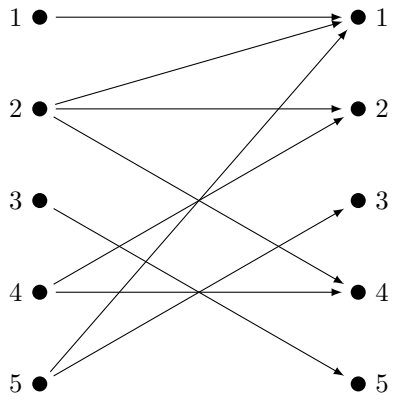


Figure 1: An example of a computational problem in graph

3 Algorithm

If I have a question of **does a pair of students in the classroom have the same birthday?** How would I solve this problem computationally? We want to create an algorithm that can be applied to any classroom, or even group of people. So the question have an arbitrary sized input.

But what exactly is an algorithm?
Algorithms are a little bit different than problems. An algorithm is like a mathematical function that takes an input and maps to a single output. It solves the problem if it generates the correct output.

Now we want to design an algorithm that solves the birthday problem.

1. Maintain a record of birthdays
2. Interview each student in some order
    - Check if birthday is in the record

- If so return the pair
- Otherwise add the birthday to the record

3. If we have interviewed everyone and there wasn't a match, then return none.

This is just the description of what an algorithm looks like. Maybe this is not enough for the computer to understand what to do, if you said this to another person, they would probably understand. **Basically an algorithm is like a recipe, it has the specific steps, or procedures that shows how to solve a problem.**

4 Correctness

If we want to argue that something is correct, we would want to prove it mathematically. In this case, we could use induction to prove the correctness of this algorithm.

In induction, we would need a base case, and some sort of predicate and a hypothesis or statement that we need to maintain. Then we would need an inductive step, which basically says I take a small value, I use the inductive step to argue it for a large number of sets.

For this algorithm, we want to prove that at the end of the interview, the algorithm has already returned the pair of match, or if there wasn't a pair, it returns none.

Proof.
Inductive Hypothesis: If first k students contain a match, then algorithm returns a match before inverviewing student $k+1$
Base Case: Let $k = 0$, then the inductive hypothesis is true because I haven't interviewed any student.
Assume inductive hypothesis is true for some $k = k'$, then we would have two cases:

1. If k' contains a match $\rightarrow$ Already returned by induction
2. Else if $k'+1$ student contains a match, the algorithm check against all the students to see if there's a match (Essentially brute force)

If there is a match, we will return it, otherwise, we have re-established the inductive hypothesis again with $k'+1$ students. Thus it completes the proof for correctness.

□

5 Efficiency

Efficiency of an algorithm does not just mean how fast the algorithm runs, but also how does it perform compared to other ways of solving the same problem.

How could we measure how fast an algorithm run? It's difficult to compare it in time, because it depends on the size of input, and also the strength of the computer that's performing the task (i.e. A smart watch vs A super computer)

Assume each kind of fundamental operation that the computers can do, take some fixed amount of time. Then we want to measure it in how many of these fundamental operations would an algorithm take to solve a problem. Which in another word is, we want to measure efficiency by counting **steps**.

6 Asymptotic Analysis

We expect performance to depend on size of our input (n).

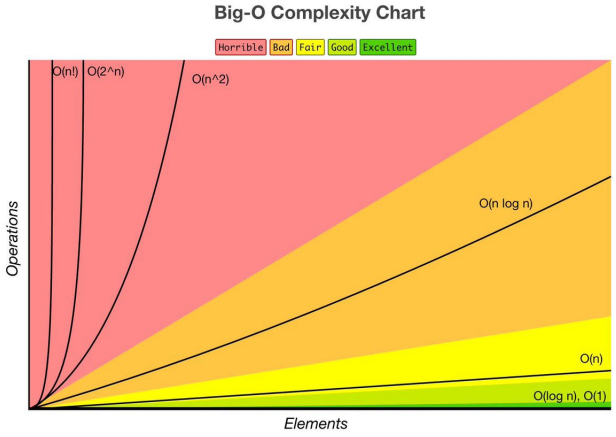
Asymptotic Notation

- $O(\dots)$ means the upper bound.
  - $g(x) \in O(f(x))$ means $\exists c, n_0 \in \mathbb{R}$ s.t. $\forall n \in \mathbb{N}, n \geq n_0 \implies g(x) \leq c \cdot f(x)$
  - In another word as input size gets bigger, $g(x)$ does now grow faster than $f(x)$ up to a constant factor.
- $\Omega(\dots)$ means the lower bound.
  - $g(x) \in \Omega(f(x))$ means $\exists c, n_0 \in \mathbb{R}$ s.t. $\forall n \in \mathbb{N}, n \geq n_0 \implies g(x) \geq c \cdot f(x)$
  - In another word as input size gets bigger, $g(x)$ grows faster than $f(x)$ up to a constant factor.
- $\Theta(\dots)$ means the tight bound (Both upper and lower bound).
  - $g(x) \in \Theta(f(x))$ means $\exists c_0, c_1, n_0 \in \mathbb{R}$ s.t. $\forall n \in \mathbb{N}, n \geq n_0 \implies c_0 \cdot f(x) \leq g(x) \leq c_1 \cdot f(x)$
  - In another word as input size gets bigger, $g(x)$ grow between $f(x)$ up to a constant factor.

Performance Order

Let n be the size of input, this is ordered from most efficient to worst.

1. $\Theta(1)$: Constant time, the amount of time it takes does not depend on the input size.
2. $\Theta(log(n))$: Logarithmic time
3. $\Theta(n)$: Linear time
4. $\Theta(n \cdot log(n))$: Log linear time
5. $\Theta(n^c)$: Polynomial time
   Up to here is what we mean by efficient in the class.
6. $\Theta(2^n)$: Exponential Time
7. $\Theta(n!)$: Factorial time



7 Model of Computation

What are computers allowed to do in constant time?
First of all, what does RAM stands for? RAM stands for **random access memory**, it means I can randomly access different places in memory in constant time.
What our model of computer is, is that you have:

1. Memory, and it's essentially just a string of bits, which are just bunch of 0's and 1's
2. CPU, which holds small information and does operations with those information. Basically it has instructions to randomly access different places in the memory, act on it and bring it back to the memory.
3. Byte is what modern computers address in, which is collection of 8 bits, so there is an address of every 8 bits in memory, so if I want to pull some information and bring it into the CPU, I give it an address, and it will take the small chunk into the CPU, operate on it and bring it back.

But how big is a chunk? It's some fixed amount of bits, which we call a **word**. A word is how big of a chunk that the CPU can take in from memory and operate at a time.
In modern computers, how big is that word size? **64 bits**, but it used to be 32 bits. There was actually a lot of issues with only 32 bits because in order to read from memory, I actually need to store that address in the CPU, in a word, and if we only have 32 bits, we only have 2^{32} different addresses. Then the biggest size of a hard disk is only 4GB. With 64 bits, we can have 2^{64} different addresses, and it turns out to be around 20 exabyte, and we are not going to run out of this limitation soon.

What are operations that I can do on a CPU

1. I can compare two words in memory
2. Integer arithmetic
3. Logical operations
4. Bitwise operations
5. Read and write from an address in memory

8 Data Structure

How can we store a large amount of data and do operations on them?

Some of the common data structures that we have seen is list, set, and dictionary. But there is a lot of code between us and the computer so it is unclear how much time that interface is taking to perform an operation.

So in the future we will talk about ways of storing a non-constant amount of information to make operations on them faster.