Counting is the the task of finding the number of elements (also called the *cardinality*) of a given set. When the set is small, we can count its elements “by hand”. When sets are larger we need a more systematic way to count.

Say you have a six-sided die and a two-sided coin — it comes out heads (`H`) or tails (`T`). What is the number of possible outcomes when both the die and the coin are tossed? There are 6 possible outcomes for the die and 2 for the coin, so the total number of outcomes is $6 \times 2 = 12$.

It will be useful to describe this kind of problem using the language of sets. The set S of possible outcomes of the die is $S = \{1, 2, 3, 4, 5, 6\}$, so $|S| = 6$. The set T of possible outcomes of the coin is $T = \{\mathtt{H}, \mathtt{T}\}$, so $|T| = 2$. The set of possible outcomes of the die *and* the coin is the *product set* $S \times T$:

$$S \times T = \{(1, \mathtt{H}), (1, \mathtt{T}), (2, \mathtt{H}), (2, \mathtt{T}), \dots, (6, \mathtt{H}), (6, \mathtt{T})\}.$$

The number of elements of $S \times T$ is $|S| \cdot |T| = 6 \cdot 2 = 12$.

In general, given any two finite sets S and T the *product set* $S \times T$ consists of all ordered pairs of elements (s, t) such that s is in S and t is in T:

$$S \times T = \{(s, t) \colon s \in S \text{ AND } t \in T\}.$$

The number of elements of $S \times T$ is the product of the number of elements of S and the number of elements of T, i.e., $|S \times T| = |S| \cdot |T|$.

Let’s do another example: Let R and B be the sets of outcomes of a toss of a red and a blue six-sided die, respectively. Then $R = \{1, 2, 3, 4, 5, 6\}$ and $B = \{1, 2, 3, 4, 5, 6\}$. When both dies are tossed, the set of outcomes is

$$R \times B = \{(1, 1), (1, 2), \dots, (6, 6)\}$$

and the number of outcomes is $|R \times B| = |R| \cdot |B| = 36$.

In cases like this when $S = T$ we can denote the set $S \times T$ by S^2. (This is the square of a set, not the square of a number). The set S^2 has $|S|^2$ elements.

We can also take the product of more than two sets. The set $S_1 \times \cdots \times S_n$ (where $S_1, \dots, S_n$ are finite sets) consists of all *sequences*[1] $(s_1, \dots, s_n)$ where s_i is in S_i for all i between 1 and n:

$$S_1 \times \cdots \times S_n = \{(s_1, \dots, s_n) \colon s_1 \in S_1 \text{ AND } \dots \text{ AND } s_n \in S_n\}.$$

The set $S_1 \times \cdots \times S_n$ has $|S_1| \cdots |S_n|$ elements.

When $S_1 = \cdots = S_n = S$, we write S^n for $S_1 \times \cdots \times S_n$. This set has $|S|^n$ elements.

For example, the set of outcomes when 9 different six-sided dies are tossed is $\{1, 2, 3, 4, 5, 6\}^9$. This set has 6^9 elements, so there are 6^9 possible outcomes.

1 Functions, bijections, and counting

One technique for counting the number of elements of a set S is to come up with a “nice” correspondence between a set S and another set T whose cardinality we already know.

[1]The order of elements in a sequence matters and there can be repetitions: For example, $(1, 1, 2)$, $(2, 1, 1)$, and $(1, 2, 2)$ are all different sequences.

Let's count the number of subsets of the set $S_n = \{1, 2, 3, \dots, n\}$.

For example, when $n = 1$, $S_1 = \{1\}$ are there are two subsets: $\varnothing$ and $\{1\}$. When $n = 2$, $S_2 = \{1, 2\}$ and there are four subsets: $\varnothing$, $\{1\}$, $\{2\}$, and $\{1, 2\}$.

I suspect that most of you can come up with a proof by induction that shows the set S_n has 2^n elements. Instead of using induction, let me show you another way.

The set $\{0, 1\}^n$ consists of all possible n-bit sequences. This is a product set, so it has 2^n elements. We will show how to represent subsets of S_n by elements of $\{0, 1\}^n$ in a unique way: Each subset of S_n is represented by an n bit sequence, and each n bit sequence represents some subset. So their number must be the same.

Here is how the representation works: The subset $\{s_1, \dots, s_k\}$ of S_n is represented by the bit sequence that has ones in positions $s_1, \dots, s_k$ and zeros in all the other positions. For example, if $n = 7$, the subset $\{3, 4, 6\}$ of S_n is represented by the bit sequence $(0, 0, 1, 1, 0, 1, 0)$ in $\{0, 1\}^n$.

Clearly every subset of S_n is represented by some n bit sequence. It is also true that every n bit sequence represents a subset: The sequence $(b_1, \dots, b_n)$ represents the set of all i between 1 and n such that $b_i = 1$. For example, the sequence $(0, 0, 1, 1, 0, 1, 1)$ represents the set $\{3, 4, 6, 7\}$. We showed a "one-to-one" correspondence between the subsets of S_n and the elements of $\{0, 1\}^n$ (n-bit sequences), so their number must be the same.

As this is our first argument of this type, let us explicitly write out the correspondence between the elements of S_2 and the sequences in $\{0, 1\}^2$:

$$\varnothing \leftrightarrow (0, 0) \qquad \{1\} \leftrightarrow (1, 0) \qquad \{2\} \leftrightarrow (0, 1) \qquad \{1, 2\} \leftrightarrow (1, 1).$$

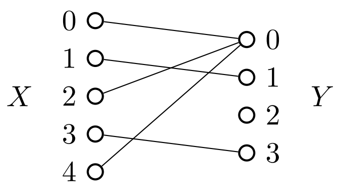
It is useful to have a language to describe these correspondences between sets. For this we need to talk about functions.

Functions A *function* f from a set X to a set Y associates to every element x in X an element $f(x)$ in Y. For example, $f(0) = 0$, $f(1) = 1$, $f(2) = 0$, $f(3) = 3$, $f(4) = 0$ is a function from set $X = \{0, 1, 2, 3, 4\}$ to the set $Y = \{0, 1, 2, 3\}$.

One way to specify a function is to list the values on all elements x in X:

x:	0	1	2	3	4
$f(x)$:	0	1	0	3	0

or by means of a bipartite graph G_f whose vertices[2] are partitioned into X and Y and whose edges are those $\{x, y\}$ such that $f(x) = y$:



A more convenient way is to describe the function using logic. This can be done in many ways, for

[2] We abuse notation here and use the same label for vertices in X and vertices in Y.

example by giving a formula for calculating f:

$$f(x) = \begin{cases} x, & \text{if } x \text{ is odd,} \\ 0, & \text{if } x \text{ is even} \end{cases} \qquad x \in \{0,1,2,3,4\}$$

an algorithm for calculating f:

```
Function $f(x)$, where $x \in \{0,1,2,3,4\}$:
    Let $b = x \bmod 2$.
    Output $b \cdot x$.
```

or, say, by giving a recurrence:

$$f(x) = (x \bmod 2) \cdot (f(x-2) + 2) \text{ for } x \geq 2, \qquad f(0) = 0, f(1) = 1.$$

No matter which way you describe a function, you must make sure that $f(x)$ is *well-defined* for every x in X: This value is specified and it is specified uniquely. For example, here is a *bad* way to define a function:

$$f(x) = y \qquad \text{where } y^2 = x.$$

this specification is consistent with both $f(1) = 1$ and $f(1) = -1$, so it does not uniquely describe the value $f(1)$.

We write $f\colon X \to Y$ for "a function from set X to set Y".

A function $f\colon X \to Y$ is a *injective* if distinct elements in x are mapped to distinct elements in Y. More precisely, f is injective if for every pair of elements x and x' in X such that $x \neq x'$, we have $f(x) \neq f(x')$. The above function is not injective because $0 \neq 2$ but $f(0) = f(2)$.

A function $f\colon X \to Y$ is *surjective* if every element y in Y is mapped to by some x in X. More precisely, f is surjective if for every y in Y there exists x in X such that $f(x) = y$. The above function is not surjective because $2 \in Y$ but nothing maps to it.

Let's assume X and Y are finite sets. When f is injective, the edges of the graph G_f are a matching and they match all the vertices of X, so Y must be at least as large as X, that is $|Y| \geq |X|$. When f is surjective, then every element of Y can be matched to some element of X (by taking an arbitrary edge incident to it), so Y can be at most as large as X, that is $|Y| \leq |X|$.

We say f is *bijective* if it is injective and surjective. When X, Y are finite and f is bijective, the edges of G_f form a perfect matching between X and Y, so $|X| = |Y|$.

This is why bijective functions are useful for counting: If we know $|X|$ and can come up with a bijective $f\colon X \to Y$, then we immediately get that $|Y| = |X|$. The tricky part is coming up with f and showing that it is bijective — namely, both injective and surjective.

Let's rework our previous example in this language. Let $X = \{0,1\}^n$ and Y be the set of all subsets of $\{1,\dots,n\}$. Let $f\colon X \to Y$ be the function that on input $(b_1,\dots,b_n)$ outputs the set of indices i such that $b_i = 1$, namely

$$f((b_1,\dots,b_n)) = \{i\colon b_i = 1\}.$$

Theorem 1. *f is a bijective function.*

Proof. First we prove f is injective: Different bit sequences map to different sets. Suppose two bit sequences x and x' are different. Then they must differ in one of their positions, say position i. Then the element i is in one of the sets $f(x), f(x')$ but not in the other, so $f(x) \neq f(x')$.

Now we show f is surjective. Given any subset S of Y, we'll show there exists a bit sequence x such that $f(x) = S$. Let x be the n-bit sequence that has a 1 in position i if i is in S and 0 if i is not in S. Then $f(x) = S$. □

Corollary 2. *The number of subsets of* $\{1, \dots, n\}$ *is* 2^n.

Proof. By Theorem 1, there is a bijection from elements of $\{0, 1\}^n$ to subsets of $\{1, \dots, n\}$. Therefore the number of subsets of $\{1, \dots, n\}$ equals the number of elements of $\{0, 1\}^n$, which is 2^n. □

2 The sum rule

The sum rule says that if $A_1, \dots, A_n$ are *disjoint* sets then

$$|A_1 \cup \cdots \cup A_n| = |A_1| + \cdots + |A_n|.$$

This rule is useful when the set whose number of elements we want to count can be written as a disjoint union of simpler sets.

For example, if you have 10 red balls, 7 blue balls, and 4 red balls, then the total number of balls you have is 10 + 7 + 4 = 21. You could have done this in first grade. So why do we need sets and unions? Sets help us break complicated counting problems into simpler ones.

Suppose you need to choose a password between 6 and 8 symbols out of which the first symbol must be a letter (lowercase or uppercase) and the remaining ones must be letters or digits. How many possible passwords are there?

Let P be the set of possible passwords, L be the set of letters, and S be the set of symbols (letters or digits). The English alphabet has 26 letters, each of which can be lowercase or uppercase, so $|L| = 52$. The set S contains all the letters plus the ten digits, so $|S| = 62$.

To count the number of passwords $|P|$, it makes sense to "decompose" the set P in terms of the simpler sets L and S. Here is how we do it. First, P is a disjoint union of six, seven, and eight letter passwords — let's denote these sets by P_6, P_7, and P_8. Therefore

$$|P| = |P_6| + |P_7| + |P_8|.$$

The set P_6 is a product set: It consists of all sequences of a letter followed by five symbols, so $P_6 = L \times S \times S \times S \times S \times S = L \times S^5$. Similarly, $P_7 = L \times S^6$ and $P_8 = L \times S_7$. Therefore

$$\begin{aligned} |P| &= |L \times S^5| + |L \times S^6| + |L \times S^7| \\ &= |L| \cdot |S|^5 + |L| \cdot |S|^6 + |L| \cdot |S|^7 \\ &= 52 \cdot 62^5 + 52 \cdot 62^6 + 52 \cdot 62^7 \\ &\approx 1.8 \cdot 10^{14}. \end{aligned}$$

The sum rule can also be used backwards as in the following example. You toss a blue six-sided die and a red six-sided die. How many outcomes are there in which the face values of the two dice come out different? Here, the set A of all possible outcomes for the pair of dice can be written as a disjoint union of the set D of outcomes in which the two face values are different and the set S in which the two are the same. By the sum rule, $|A| = |D| + |S|$. Since $|A| = 36$ and $|S| = 6$, we get that $|D| = 36 - 6 = 30$ possible outcomes in which the two face values are different.

3 The general product rule and permutations

Let's see a different solution to the last example. The set of possible outcomes in which the two face values are different is *not* a product set, but we can still reason like this: There are 6 possibilities for the face value of the first die. For each one of these six possibilities, we are interested in the outcomes in which the second die has a different face value. No matter what the toss of the first die came out to be, there are always five possible choices for the toss of the second die that make their face values different. Therefore the total number of outcomes is $6 \times 5 = 30$.

This is an instance of the *general product rule*: Given a set S of sequences of length k, where

- There are n_1 possible first entries,
- There are n_2 possible second entries for each first entry
- There are n_3 possible third entries for each combination of first and second entries, and so on up to n_k

the size of S is $n_1 \cdot n_2 \cdots n_k$.

A *permutation* of a set S is a sequence that contains every element of S exactly once. For example, the permutations of the set $\{1, 2, 3\}$ are the sequences $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$, $(3, 2, 1)$.

How many permutations does an n element set have? The first entry in the permutation sequence can be chosen in n possible ways. Each such choice leaves out $n - 1$ possibilities for the second element. Each combination of choices for the first two elements leaves out $n - 2$ combinations for the third element. Continuing like this, we get that the number of permutations of an n element set is

$$n \cdot (n-1) \cdot (n-2) \cdots 1 = n!.$$

Let's do another example of the product rule. In how many ways can you place three different pieces on an 8×8 chessboard — a bishop, a knight, and a pawn — so that no two pieces share a row or a column? The position of the three pieces is specified by a six numbers (br, bc, kr, kc, pr, pc). The first number br indicates the bishop's row, the second number bc indicates the bishop's column, and so on.

We can count the number of allowed sequences with the generalized product rule. There are 8 possibilities for the bishop's row br. For each such choice, there are 8 possibilities for the bishop's column bc. Once the bishop's position was chosen, there are 7 choices for the knight's row kr (any one but br) and (for each of them) 7 choices from kc (any one but bc. Once the bishop and the knight were positioned, there are 6 choices for pr and (for each of them) 6 choices of pc. The total number of configurations is $8 \cdot 8 \cdot 7 \cdot 7 \cdot 6 \cdot 6 = (8 \cdot 7 \cdot 6)^2$.

4 The pigeonhole principle

The pigeonhole principle says that if you toss more pigeons into fewer holes, at least two pigeons will land in the same hole. If X is the set of pigeons, Y is the set of holes, and $f \colon X \to Y$ is the function that tells you which pigeon goes into which hole, we get the following mathematical statement:

Theorem 3 (The pigeonhole principle). *For any two finite sets X and Y such that $|X| > |Y|$ and any function $f \colon X \to Y$, there exist inputs $x \neq x'$ such that $f(x) = f(x')$.*

Proof. We prove the contrapositive. If for all inputs $x \neq x'$, $f(x) \neq f(x')$, then f is injective so $|Y| \geq |X|$. □

The pigeonhole principle quickly answers questions that may otherwise look impossibly difficult. Let's do a few examples.

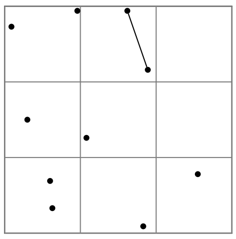
Example 1. In a room of 400 people, there must be two that have the same birthday.

Here, X is the set of people in the room, Y is the set of 365 days in the year, and $f(x) = y$ if person x was born on day y of the year. By the pigeonhole principle, there must be a pair of people $x \neq x'$ such that $f(x) = f(x')$, that is they are born on the same day of the year.

Example 2. Among any 10 points in a unit (1×1) square, there must be a pair that are within distance $\sqrt{2}/3 \approx 0.471$.

Divide the unit square evenly into nine $\frac{1}{3} \times \frac{1}{3}$ blocks. Let X be the set of ten points, Y be the set of nine blocks, and $f(x) = y$ if point x falls inside block y. By the pigeonhole principle, at least two points must fall into the same block. The diameter of this block is $\frac{1}{3}\sqrt{2}$, so the distance between the two points within the block cannot be larger than this number.

Here is an example of a possible configuration. The two vertices connected by a line fall within the same block, so they must be within a distance of $\frac{1}{3}\sqrt{2}$.



Example 3. You have a set A of 8 integers, each of them between 0 and 30. There are two different subsets of S so that the sum of the numbers in each subset is the same.

Let X be the set of all subsets of A, Y be the set $\{0, 1, \dots, 240\}$, and $f \colon X \to Y$ be the function that on input S, outputs the sum of all integers in S. Since there are 8 numbers between 0 and 30, this sum is always a number between 0 and $8 \cdot 30 = 240$.

The number of subsets of A is $2^8 = 256$. On the other hand, $|Y| = 241$. By the pigeonhole principle, two different subsets S_1, S_2 of A must have the same sum.

We can even make the stronger conclusion that there exists two *disjoint* subsets of S with the same sum: If S_1 and S_2 are not disjoint, take away the elements in their intersection from both of them. This changes the value of both sums by the same amount, so the sums stay equal but the sets are now disjoint.

There is a more general variant of the pigeonhole principle that is sometimes useful.

Theorem 4 (Generalized pigeonhole principle)**.** *For every integer k every pair of sets X, Y such that $|X| > k|Y|$ and every function $f \colon X \to Y$, there exist $k + 1$ distinct elements $x_1, \dots, x_{k+1}$ of X such that $f(x_1) = \cdots = f(x_{k+1})$.*

Proof. We prove the contrapositive. Suppose that for every $y \in Y$, there are at most k distinct elements of x that map to it. Order the elements of X in some way from smallest to largest. Let $f'\colon X \to Y \times \{1,\dots,k\}$ be the function such that $f'(x) = (y, i)$ if x is the i-th smallest element of X among those that $f(x) = y$. Then the function f is injective, so $|Y \times \{1,\dots,k\}| \geq |X|$. By the product rule, $|Y \times \{1,\dots,k\}| = k|Y|$ and so $k|Y| \geq |X|$. □

Example 4. In a group of 1500 people, there must be three people of the same gender that have the same birthday.

Since $1500 > 4 \cdot 365$, by the generalized pigeonhole principle there must be at least five people in the group that have the same birthday. Let X be these five people, and $f\colon X \to \{\text{male}, \text{female}\}$ be the map that assigns each person their gender. By the generalized pigeonhole principle, there are at least three people among the five that have the same gender.

Here is another solution. Let X be the group of 1500 people, Y be the product set $\{1,\dots,365\} \times \{\text{male}, \text{female}\}$, and $g\colon X \to Y$ be the function that assigns to each person x in the group their birthday and their gender. Since $|Y| = 365 \cdot 2 = 730$ and $1500 > 2 \cdot 730$, by the generalized pigeonhole principle there are three people in the group with the same birthday and the same gender.

5 Comparison based sorting

In Lecture 8 we showed that the Merge Sort procedure makes $n \log n - n + 1$ pairwise comparisons when sorting any sequence of length n (assuming n is a power of two). Is there a different procedure that performs fewer comparisons?

We will argue that any procedure for sorting a sequence of numbers that only relies on comparing pairs of numbers (and not on other things, like the values of these numbers) cannot do substantially better. No matter what the procedure does, there exists at least one sequence of length n on which the procedure performs $\Omega(n \log n)$ comparisons.

How can we even reason about a sorting procedure without knowing how it works? The main difficulty here is to come up with a good *model* of sorting procedures. Once we do that, the calculations won't be hard. Instead of sorting, let's first talk about a simpler but related example — the game of twenty questions.

Twenty questions The game of twenty questions involves two players, Alice and Bob. Alice thinks of an integer between 0 and $N - 1$ and writes it on a piece of paper. Bob and Alice then alternate in asking and answering questions about this number. Bob can ask any question as long as it has a yes/no answer and Alice always answers Bob's questions truthfully. How many questions does Bob need to ask in order to find out Alice's number with certainty?

Here is a simple strategy for Bob, assuming N is a power of two, say $N = 2^q$. Represent each integer between 0 and $N-1$ uniquely by its base 2 expansion $(b_1, \dots, b_q)$.[3] For each i ranging from 1 to q, ask the question "Is $b_i = 1$"? Alice's answer to this question determines the value of b_i (if yes $b_i = 1$, if no $b_i = 0$) so after q questions Bob finds the binary representation of Alice's secret. With this strategy, Bob asks $q = \log N$ questions.

[3]The binary-to-integer conversion $f((b_1, \dots, b_n)) = b_1 + 2b_2 + \cdots + 2^{q-1}b_{q-1}$ is a bijective function from $\{0,1\}^n$ to the set $\{0, \dots, N-1\}$, so each number $x \in \{1, \dots, N\}$ is uniquely represented by a bit sequence $(b_1, \dots, b_q)$ such that $f((b_1, \dots, b_q)) = x$.

Let us now argue that no matter which strategy Bob uses, there is at least one possible secret for Alice that makes Bob ask at least $\log N$ questions. Assume, for contradiction, that Bob can guess Alice's secret with certainty after asking $q < \log N$ questions. If we represent the answers to Bob's questions by a $\{\text{yes}, \text{no}\}$ sequence of length q, then the number of possible answer sequences is $|\{\text{yes}, \text{no}\}^q| = 2^q$. As there are N possible secrets for Alice and $N > 2^q$, by the pigeonhole principle at least two of Alice's possible secrets would result in the same sequence of answers to Bob's questions. After asking q questions and seeing these answers, Bob will not know which of these two is Alice's real secret so he cannot guess it with certainty.

We model the execution of a comparison-based sorting procedure as a "twenty questions" game between the input sequence (Alice) and the sorting procedure (Bob). The procedure only asks questions of the type "is entry i in the sequence smaller than entry j?" which have yes/no answers.

Let's do a small example. Suppose we have some sorting procedure — which we call Bob — that was given the sequence (a_1, a_2, a_3) to sort. We'll assume a_1, a_2, a_3 are all distinct. Initially, Bob does not know anything about this sequence so he asks a question like "is $a_1 < a_2$?" Say the answer to this question is "yes". Then he asks for example "Is $a_2 < a_3$?" If the answer to this question is also "yes", then Bob has enough information to determine the correct order of the sequence, namely $a_1 < a_2 < a_3$.

Now suppose to the first question "Is $a_1 < a_2$" is "yes" but the answer to "Is $a_2 < a_3$?" is "no". There are two possible orderings of the sequence that are consistent with these answers: $a_1 < a_2 < a_3$ and $a_2 < a_1 < a_3$. Bob cannot sort the sequence just based on this information.

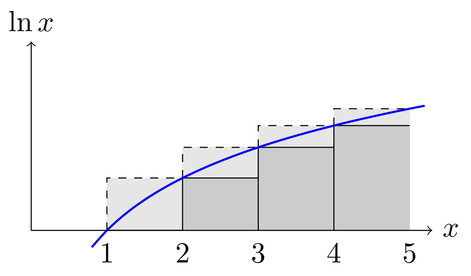
Let's fix $a_1, \dots, a_n$ to be any n distinct values to sort and let S be the set of all *permutations* of $\{a_1, \dots, a_n\}$. Bob could be given any of these permutations as an input sequence. The answers to his first q comparison questions are $\{\text{yes}, \text{no}\}$ sequences of length q. If $|S| > 2^q$, then by the pigeonhole principle at least two input sequences lead to the same sequence of answers. Bob cannot distinguish which one of these two is his input, and so he cannot sort it properly.

We conclude that if S has strictly more than 2^q elements, then not all sequences of n elements can be sorted after q comparisons. The set S consists of the permutations of $\{a_1, \dots, a_n\}$, so it has size $n!$. Sorting with q comparisons is impossible as long as $n! > 2^q$; in other words sorting *requires* at least $\log n!$ comparisons.

What does the number $\log_2 n!$ look like? We can write

$$\log n! = \frac{\ln n!}{\ln 2} = \frac{1}{\ln 2}\ln(1 \cdot 2 \cdots n) = \frac{1}{\ln 2}(\ln 1 + \ln 2 + \cdots + \ln n).$$

This sum is perfect for the integral method.



Lower bounding the sum by the related integral minus the error term gives

$$\ln 1 + \ln 2 + \cdots + \ln n \geq \int_1^{n+1} \ln x \, dx - \ln(n+1)$$

The antiderivative of $\ln x$ is $x \ln x - x$, so

$$\ln 1 + \ln 2 + \cdots + \ln n \geq (n+1)\ln(n+1) - n - \ln(n+1).$$

Therefore any procedure for comparison-based sorting requires at least

$$(n+1)\log(n+1) - \frac{n}{\ln 2} - \log(n+1) = \Omega(n \log n)$$

comparisons on at least one input sequence.

6 Data integrity*

Bob has downloaded a piece of software from that was supposedly produced by trusted Alice from an internet software provider. He is not sure that the provider is legitimate and are afraid that it may install a virus on your computer instead. How can Bob verify the integrity of the file?

A collision-resistant hash function is a special type of function that can be used to certify data integrity. A *hash function* is a function h that takes as an input an arbitrarily long string x (representing some data like a piece of software) and outputs a short (say, 1000-bit) certificate of integrity $h(x)$. The hash function is *collision-resistant* if it is infeasible to find two distinct inputs with the same certificate, namely two strings $x \neq x'$ such that $h(x) = h(x')$. Such a pair of strings is said to form a *collision* under h.

Armed with a collision-resistant hash function h, Bob can verify the integrity of his download x' by computing the value $h(x')$ and checking that it matches the value of $h(x)$ for the original file x. Alice can, for instance, post the short string $h(x)$ on her web page for this purpose. For a malicious software provider to pass Bob's test, it must produce a corrupted copy x' that collides with the original x under h, which is an infeasible task.

How does one construct a collision-resistant hash function? There are several proposals that appear to work, although we do not know how to prove that any of them are secure with respect to finding collisions. I will describe a particularly elegant construction based on modular arithmetic.

Modular subset sum To describe the hash function we need a few concepts from modular linear algebra. An n-dimensional vector modulo q is a sequence of n numbers in the range $\{0, \ldots, q-1\}$, each representing a possible residue modulo q. Vectors of the same dimension can be added elementwise using the rules of modular arithmetic, for instance

$$\begin{bmatrix}3\\6\\1\end{bmatrix} + \begin{bmatrix}5\\7\\0\end{bmatrix} \bmod 8 = \begin{bmatrix}0\\5\\1\end{bmatrix}$$

where we use the standard convention of representing vectors as columns.

The modular subset sum function is a function whose inputs are subsets of the set $\{1, \ldots, m\}$ and whose outputs are n-dimensional vectors modulo q. We first fix a collection of m n-dimensional vectors modulo q, which we denote by $\mathbf{a}_1$ up to $\mathbf{a}_m$. These vectors are chosen independently at random among all q^n possibilities. The modular subset sum function applied to a set S is the modular sum of the vectors indexed by S, namely

$$h(S) = \sum_{i \in S} \mathbf{a}_i \bmod q.$$

For example, when $m = 10, n = 3, q = 8$, we may have

$$\begin{bmatrix} 3 & 1 & 1 & 6 & 1 & 3 & 0 & 0 & 1 & 5 \\ 3 & 4 & 5 & 2 & 3 & 6 & 0 & 6 & 3 & 2 \\ 4 & 4 & 1 & 2 & 2 & 3 & 2 & 2 & 6 & 6 \end{bmatrix}$$

with $\mathbf{a}_i$ being the i-th column of this matrix. Then

$$h(\{2,3,5,7\}) = \begin{bmatrix} 1 \\ 4 \\ 4 \end{bmatrix} + \begin{bmatrix} 1 \\ 5 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix} \bmod 8 = \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix}. \tag{1}$$

In general, the domain of h consists of all subsets of $\{1, \dots, m\}$, of which there are 2^m. Each subset can be identified with an m-bit string through the bijection in Theorem 1. The range of h consists of all n-dimensional vectors modulo q, of which there are q^n. If q is a power of two and we represent each number modulo q by its base 2 representation, then the output can be viewed as a bit string of length $n \log q$ obtained by concatenating the bit string representations of all the vector entries. Under this representation, the function h in our example can be viewed as a function from 10-bit strings to 9-bit strings. In this notation, (1) can be rewritten as

$$h(0110101000) = 011\ 100\ 001.$$

By the pigeonhole principle, h must contain a collision. More generally, when $m > n \log q$, the pigeonhole principle guarantees that h must contain a collision. How easy is it to find this collision?

One possibility is to carry out exhaustive search over all possible subsets. For example, by trying out all possible 2^{10} subsets S in the above example I found that the set $\{1, 2, 3, 6, 7, 8\}$ collides with the empty set, namely

$$h(\{1,2,3,6,7,8\}) = h(\varnothing) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

For larger values of m, like $m = 1,536$, exhaustive search over all 2^m possible subsets becomes prohibitively expensive. A more efficient strategy is to try random sets until we find a pair that collides. It turns out that in order to observe a collision with good probability one needs to try out about $\sqrt{q^n} = 2^{(n \log q)/2}$ sets (see the birthday paradox). This is an improvement, but still quite costly when, say, $n = 64$ and $q = 4,096$. There are more sophisticated methods for finding collisions, but for this setting of parameters even the best known ones may need to try out something on the order of 2^{100} different inputs, which is infeasible. It seems therefore reasonable to conjecture that

> It is infeasible to find collisions in the subset sum function h with parameters $m = 1,536$, $n = 64$, and $q = 4,096$.

Viewed as a function between bit strings, h hashes strings of length $m = 1,536$ into strings of length $n \log q = 768$, so h shrinks the size of its input by half. In terms of sets, the domain and range of h have size $2^{1,536}$ and 2^{768}, respectively, so by the generalized pigeonhole principle h must have at least 2^{768} distinct inputs that map to the same output. Nonetheless, even a single pair of such inputs is extremely difficult to find!

The hash function h we just described is presumably collision-resistant, but it can only be applied to inputs that are 1,536 bits long. How can we use it to certify the integrity of a data file x that could be several hundreds of megabytes long? One solution is to break x into chunks $x_1 x_2 \dots x_\ell$, each of them 768-bits long, and to apply h iteratively on the chunks like this:

$$H(x) = h(\cdots h(h(h(x_1 x_2) x_3) x_4) \cdots x_\ell).$$

This is a legitimate way to compose h, as each copy takes two inputs of length 768 (a total of of 1,536 input bits) and produces an output of length 768.

We claim that if h is collision-resistant, so is H. For concreteness, let's work out the case $\ell = 3$; the argument can easily be extended to larger values of ℓ. Suppose, for contradiction, that a malicious software provider Eve managed to find two different files $x = x_1x_2x_3$ and $x' = x'_1x'_2x'_3$ that collide under H. By the definition of H, this means

$$h(h(x_1x_2)x_3) = h(h(x'_1x'_2)x'_3)$$

There are two possibilities. One possibility is that $h(x_1x_2)x_3$ is distinct from $h(x'_1x'_2)x'_3$. In this case, Eve found the strings $h(x_1x_2)x_3$ and $h(x'_1x'_2)x'_3$ (viewed as strings of length 1,536 each) that collide under h, which is infeasible. The other possibility is that the strings $h(x_1x_2)x_3$ and $h(x'_1x'_2)x'_3$ are equal. Then $x_3 = x'_3$, so x_1x_2 and $x'_1x'_2$ must be distinct, but $h(x_1x_2)$ and $h(x'_1x'_2)$ are equal. So eve found a collision between x_1x_2 and $x'_1x'_2$ under h, which is also infeasible. In either case, the only way in which Eve can find a collision under H is by finding a collision under h, which is an infeasible problem.

References

This lecture is based on Chapters 4 and 15 of the text *Mathematics for Computer Science* by E. Lehman, T. Leighton, and A. Meyer. The discussion on collision-resistant hashing and modular subset sum is partly based on Section 4 of the paper Lattice-based Cryptography by D. Micciancio and O. Regev and on Section 4.6 of the book *Introduction to Modern Cryptography* by J. Katz and Y. Lindell.