About Chapter 12

In Chapters 1–11, we concentrated on two aspects of information theory and coding theory: source coding – the compression of information so as to make efficient use of data transmission and storage channels; and channel coding – the redundant encoding of information so as to be able to detect and correct communication errors.

In both these areas we started by ignoring practical considerations, concentrating on the question of the theoretical limitations and possibilities of coding. We then discussed practical source-coding and channel-coding schemes, shifting the emphasis towards computational feasibility. But the prime criterion for comparing encoding schemes remained the efficiency of the code in terms of the channel resources it required: the best source codes were those that achieved the greatest compression; the best channel codes were those that communicated at the highest rate with a given probability of error.

In this chapter we now shift our viewpoint a little, thinking of *ease of information retrieval* as a primary goal. It turns out that the random codes which were theoretically useful in our study of channel coding are also useful for rapid information retrieval.

Efficient information retrieval is one of the problems that brains seem to solve effortlessly, and content-addressable memory is one of the topics we will study when we look at neural networks.

12

Hash Codes: Codes for Efficient Information Retrieval

▶ 12.1 The information-retrieval problem

A simple example of an information-retrieval problem is the task of implementing a phone directory service, which, in response to a person's *name*, returns (a) a confirmation that that person is listed in the directory; and (b) the person's phone number and other details. We could formalize this problem as follows, with S being the number of names that must be stored in the directory.

string length	$N \simeq 200$
number of strings	$S \simeq 2^{23}$
number of possible strings	$2^N \simeq 2^{200}$

Figure 12.1. Cast of characters.

You are given a list of S binary strings of length N bits, $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(S)}\}$, where S is considerably smaller than the total number of possible strings, 2^N. We will call the superscript 's' in $\mathbf{x}^{(s)}$ the *record number* of the string. The idea is that s runs over customers in the order in which they are added to the directory and $\mathbf{x}^{(s)}$ is the name of customer s. We assume for simplicity that all people have names of the same length. The name length might be, say, $N = 200$ bits, and we might want to store the details of ten million customers, so $S \simeq 10^7 \simeq 2^{23}$. We will ignore the possibility that two customers have identical names.

The task is to construct the inverse of the mapping from s to $\mathbf{x}^{(s)}$, i.e., to make a system that, given a string $\mathbf{x}$, returns the value of s such that $\mathbf{x} = \mathbf{x}^{(s)}$ if one exists, and otherwise reports that no such s exists. (Once we have the record number, we can go and look in memory location s in a separate memory full of phone numbers to find the required number.) The aim, when solving this task, is to use minimal computational resources in terms of the amount of memory used to store the inverse mapping from $\mathbf{x}$ to s and the amount of time to compute the inverse mapping. And, preferably, the inverse mapping should be implemented in such a way that further new strings can be added to the directory in a small amount of computer time too.

Some standard solutions

The simplest and dumbest solutions to the information-retrieval problem are a look-up table and a raw list.

The look-up table is a piece of memory of size $2^N \log_2 S$, $\log_2 S$ being the amount of memory required to store an integer between 1 and S. In each of the 2^N locations, we put a zero, except for the locations $\mathbf{x}$ that correspond to strings $\mathbf{x}^{(s)}$, into which we write the value of s.

The look-up table is a simple and quick solution, but only if there is sufficient memory for the table, and if the cost of looking up entries in memory is independent of the memory size. But in our definition of the task, we assumed that N is about 200 bits or more, so the amount of memory required would be of size 2^{200}; this solution is completely out of the question. Bear in mind that the number of particles in the solar system is only about 2^{190}.

The raw list is a simple list of ordered pairs $(s, \mathbf{x}^{(s)})$ ordered by the value of s. The mapping from $\mathbf{x}$ to s is achieved by searching through the list of strings, starting from the top, and comparing the incoming string $\mathbf{x}$ with each record $\mathbf{x}^{(s)}$ until a match is found. This system is very easy to maintain, and uses a small amount of memory, about SN bits, but is rather slow to use, since on average five million pairwise comparisons will be made.

▷ Exercise 12.1.[*2*, p.202] Show that the average time taken to find the required string in a raw list, assuming that the original names were chosen at random, is about $S + N$ binary comparisons. (Note that you don't have to compare the whole string of length N, since a comparison can be terminated as soon as a mismatch occurs; show that you need on average two binary comparisons per incorrect string match.) Compare this with the worst-case search time – assuming that the devil chooses the set of strings and the search key.

The standard way in which phone directories are made improves on the look-up table and the raw list by using an *alphabetically-ordered list*.

Alphabetical list. The strings $\{\mathbf{x}^{(s)}\}$ are sorted into alphabetical order. Searching for an entry now usually takes less time than was needed for the raw list because we can take advantage of the sortedness; for example, we can open the phonebook at its middle page, and compare the name we find there with the target string; if the target is 'greater' than the middle string then we know that the required string, if it exists, will be found in the second half of the alphabetical directory. Otherwise, we look in the first half. By iterating this splitting-in-the-middle procedure, we can identify the target string, or establish that the string is not listed, in $\lceil \log_2 S \rceil$ string comparisons. The expected number of binary comparisons per string comparison will tend to increase as the search progresses, but the total number of binary comparisons required will be no greater than $\lceil \log_2 S \rceil N$.

The amount of memory required is the same as that required for the raw list.

Adding new strings to the database requires that we insert them in the correct location in the list. To find that location takes about $\lceil \log_2 S \rceil$ binary comparisons.

Can we improve on the well-established alphabetized list? Let us consider our task from some new viewpoints.

The task is to construct a mapping $\mathbf{x} \to s$ from N bits to $\log_2 S$ bits. This is a pseudo-invertible mapping, since for any $\mathbf{x}$ that maps to a non-zero s, the customer database contains the pair $(s, \mathbf{x}^{(s)})$ that takes us back. Where have we come across the idea of mapping from N bits to M bits before?

We encountered this idea twice: first, in source coding, we studied block codes which were mappings from strings of N symbols to a selection of one label in a list. The task of information retrieval is similar to the task (which we never actually solved) of making an encoder for a typical-set compression code.

The second time that we mapped bit strings to bit strings of another dimensionality was when we studied channel codes. There, we considered codes that mapped from K bits to N bits, with N greater than K, and we made theoretical progress using *random* codes.

In hash codes, we put together these two notions. We will study random codes that map from N bits to M bits where M is *smaller* than N.

The idea is that we will map the original high-dimensional space down into a lower-dimensional space, one in which it is feasible to implement the dumb look-up table method which we rejected a moment ago.

▶ 12.2 Hash codes

string length	$N \simeq 200$
number of strings	$S \simeq 2^{23}$
size of hash function	$M \simeq 30$ bits
size of hash table	$T = 2^M$ $\simeq 2^{30}$

Figure 12.2. Revised cast of characters.

First we will describe how a hash code works, then we will study the properties of idealized hash codes. A hash code implements a solution to the information-retrieval problem, that is, a mapping from $\mathbf{x}$ to s, with the help of a pseudo-random function called a *hash function*, which maps the N-bit string $\mathbf{x}$ to an M-bit string $\mathbf{h}(\mathbf{x})$, where M is smaller than N. M is typically chosen such that the 'table size' $T \simeq 2^M$ is a little bigger than S – say, ten times bigger. For example, if we were expecting S to be about a million, we might map $\mathbf{x}$ into a 30-bit hash $\mathbf{h}$ (regardless of the size N of each item $\mathbf{x}$). The hash function is some fixed deterministic function which should ideally be indistinguishable from a fixed random code. For practical purposes, the hash function must be quick to compute.

Two simple examples of hash functions are:

Division method. The table size T is a prime number, preferably one that is not close to a power of 2. The hash value is the remainder when the integer $\mathbf{x}$ is divided by T.

Variable string addition method. This method assumes that $\mathbf{x}$ is a string of bytes and that the table size T is 256. The characters of $\mathbf{x}$ are added, modulo 256. This hash function has the defect that it maps strings that are anagrams of each other onto the same hash.

It may be improved by putting the running total through a fixed pseudorandom permutation after each character is added. In the *variable string exclusive-or method* with table size $\leq 65\,536$, the string is hashed twice in this way, with the initial running total being set to 0 and 1 respectively (algorithm 12.3). The result is a 16-bit hash.

Having picked a hash function $\mathbf{h}(\mathbf{x})$, we implement an information retriever as follows. (See figure 12.4.)

Encoding. A piece of memory called the *hash table* is created of size $2^M b$ memory units, where b is the amount of memory needed to represent an integer between 0 and S. This table is initially set to zero throughout. Each memory $\mathbf{x}^{(s)}$ is put through the hash function, and at the location in the hash table corresponding to the resulting vector $\mathbf{h}^{(s)} = \mathbf{h}(\mathbf{x}^{(s)})$, the integer s is written – unless that entry in the hash table is already occupied, in which case we have a *collision* between $\mathbf{x}^{(s)}$ and some earlier $\mathbf{x}^{(s')}$ which both happen to have the same hash code. Collisions can be handled in various ways – we will discuss some in a moment – but first let us complete the basic picture.

```
unsigned char Rand8[256];     // This array contains a random
                                 permutation from 0..255 to 0..255
int Hash(char *x) {           // x is a pointer to the first char;
     int h;                   //    *x is the first character
     unsigned char h1, h2;

     if (*x == 0) return 0;   // Special handling of empty string
     h1 = *x; h2 = *x + 1;    // Initialize two hashes
     x++;                     // Proceed to the next character
     while (*x) {
         h1 = Rand8[h1 ^ *x]; // Exclusive-or with the two hashes
         h2 = Rand8[h2 ^ *x]; //    and put through the randomizer
         x++;
     }                        // End of string is reached when *x=0
     h = ((int)(h1)<<8) |     // Shift h1 left 8 bits and add h2
          (int) h2 ;
     return h ;               // Hash is concatenation of h1 and h2
}
```

Algorithm 12.3. C code implementing the variable string exclusive-or method to create a hash h in the range $0 \ldots 65\,535$ from a string x. Author: Thomas Niemann.

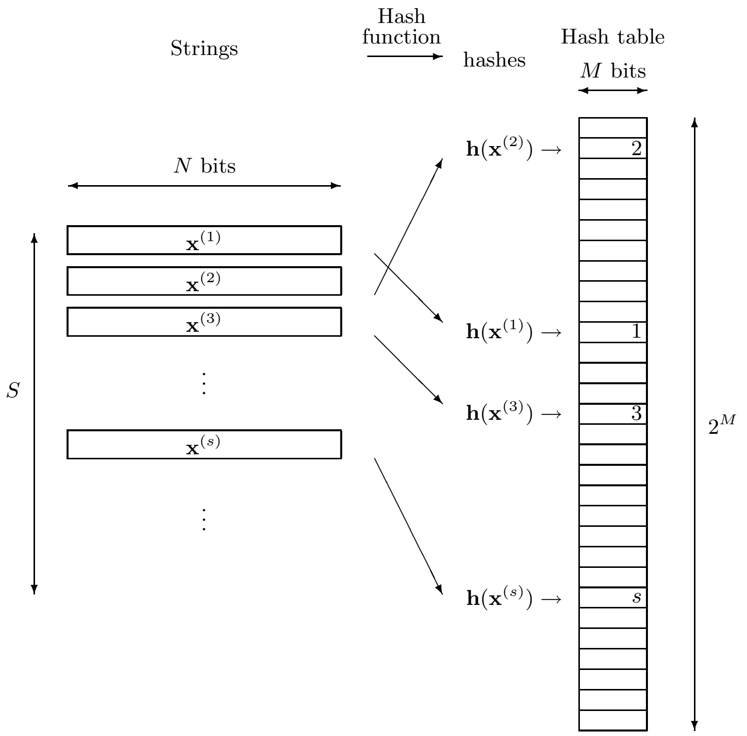
Strings
Hash function
hashes
Hash table
M bits
N bits
$\mathbf{x}^{(1)}$
$\mathbf{x}^{(2)}$
$\mathbf{x}^{(3)}$
$\mathbf{x}^{(s)}$
S
$\mathbf{h}(\mathbf{x}^{(2)}) \rightarrow$
$\mathbf{h}(\mathbf{x}^{(1)}) \rightarrow$
$\mathbf{h}(\mathbf{x}^{(3)}) \rightarrow$
$\mathbf{h}(\mathbf{x}^{(s)}) \rightarrow$
2
1
3
s
2^M

Figure 12.4. Use of hash functions for information retrieval. For each string $\mathbf{x}^{(s)}$, the hash $\mathbf{h} = \mathbf{h}(\mathbf{x}^{(s)})$ is computed, and the value of s is written into the $\mathbf{h}$th row of the hash table. Blank rows in the hash table contain the value zero. The table size is $T = 2^M$.

Decoding. To retrieve a piece of information corresponding to a target vector $\mathbf{x}$, we compute the hash $\mathbf{h}$ of $\mathbf{x}$ and look at the corresponding location in the hash table. If there is a zero, then we know immediately that the string $\mathbf{x}$ is not in the database. The cost of this answer is the cost of one hash-function evaluation and one look-up in the table of size 2^M. If, on the other hand, there is a non-zero entry s in the table, there are two possibilities: either the vector $\mathbf{x}$ is indeed equal to $\mathbf{x}^{(s)}$; or the vector $\mathbf{x}^{(s)}$ is another vector that happens to have the same hash code as the target $\mathbf{x}$. (A third possibility is that this non-zero entry might have something to do with our yet-to-be-discussed collision-resolution system.)

To check whether $\mathbf{x}$ is indeed equal to $\mathbf{x}^{(s)}$, we take the tentative answer s, look up $\mathbf{x}^{(s)}$ in the original forward database, and compare it bit by bit with $\mathbf{x}$; if it matches then we report s as the desired answer. This successful retrieval has an overall cost of one hash-function evaluation, one look-up in the table of size 2^M, another look-up in a table of size S, and N binary comparisons – which may be much cheaper than the simple solutions presented in section 12.1.

▷ Exercise 12.2.[*2*, p.202] If we have checked the first few bits of $\mathbf{x}^{(s)}$ with $\mathbf{x}$ and found them to be equal, what is the probability that the correct entry has been retrieved, if the alternative hypothesis is that $\mathbf{x}$ is actually not in the database? Assume that the original source strings are random, and the hash function is a random hash function. How many binary evaluations are needed to be sure with odds of a billion to one that the correct entry has been retrieved?

The hashing method of information retrieval can be used for strings $\mathbf{x}$ of arbitrary length, if the hash function $\mathbf{h}(\mathbf{x})$ can be applied to strings of any length.

▶ 12.3 Collision resolution

We will study two ways of resolving collisions: appending in the table, and storing elsewhere.

Appending in table

When encoding, if a collision occurs, we continue down the hash table and write the value of s into the next available location in memory that currently contains a zero. If we reach the bottom of the table before encountering a zero, we continue from the top.

When decoding, if we compute the hash code for $\mathbf{x}$ and find that the s contained in the table doesn't point to an $\mathbf{x}^{(s)}$ that matches the cue $\mathbf{x}$, we continue down the hash table until we either find an s whose $\mathbf{x}^{(s)}$ does match the cue $\mathbf{x}$, in which case we are done, or else encounter a zero, in which case we know that the cue $\mathbf{x}$ is not in the database.

For this method, it is essential that the table be substantially bigger in size than S. If $2^M < S$ then the encoding rule will become stuck with nowhere to put the last strings.

Storing elsewhere

A more robust and flexible method is to use *pointers* to additional pieces of memory in which collided strings are stored. There are many ways of doing this. As an example, we could store in location **h** in the hash table a pointer (which must be distinguishable from a valid record number s) to a 'bucket' where all the strings that have hash code **h** are stored in a *sorted list*. The encoder sorts the strings in each bucket alphabetically as the hash table and buckets are created.

The decoder simply has to go and look in the relevant bucket and then check the short list of strings that are there by a brief alphabetical search.

This method of storing the strings in buckets allows the option of making the hash table quite small, which may have practical benefits. We may make it so small that almost all strings are involved in collisions, so all buckets contain a small number of strings. It only takes a small number of binary comparisons to identify which of the strings in the bucket matches the cue $\mathbf{x}$.

▶ 12.4 Planning for collisions: a birthday problem

Exercise 12.3.[*2*, p.202] If we wish to store S entries using a hash function whose output has M bits, how many collisions should we expect to happen, assuming that our hash function is an ideal random function? What size M of hash table is needed if we would like the expected number of collisions to be smaller than 1?

What size M of hash table is needed if we would like the expected number of collisions to be a small fraction, say 1%, of S?

[Notice the similarity of this problem to exercise 9.20 (p.156).]

▶ 12.5 Other roles for hash codes

Checking arithmetic

If you wish to check an addition that was done by hand, you may find useful the method of *casting out nines*. In casting out nines, one finds the sum, modulo nine, of all the *digits* of the numbers to be summed and compares it with the sum, modulo nine, of the digits of the putative answer. [With a little practice, these sums can be computed much more rapidly than the full original addition.]

Example 12.4. In the calculation shown in the margin the sum, modulo nine, of the digits in 189+1254+238 is 7, and the sum, modulo nine, of 1+6+8+1 is 7. The calculation thus passes the casting-out-nines test.

```
  189
+1254
+ 238
-----
 1681
```

Casting out nines gives a simple example of a hash function. For any addition expression of the form $a + b + c + \cdots$, where $a, b, c, \ldots$ are decimal numbers we define $h \in \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ by

$$h(a + b + c + \cdots) = \text{ sum modulo nine of all digits in } a, b, c \ ; \qquad (12.1)$$

then it is nice property of decimal arithmetic that if

$$a + b + c + \cdots = m + n + o + \cdots \qquad (12.2)$$

then the hashes $h(a + b + c + \cdots)$ and $h(m + n + o + \cdots)$ are equal.

▷ Exercise 12.5.[*1*, p.203] What evidence does a correct casting-out-nines match give in favour of the hypothesis that the addition has been done correctly?

Error detection among friends

Are two files the same? If the files are on the same computer, we could just compare them bit by bit. But if the two files are on separate machines, it would be nice to have a way of confirming that two files are identical without having to transfer one of the files from A to B. [And even if we did transfer one of the files, we would still like a way to confirm whether it has been received without modifications!]

This problem can be solved using hash codes. Let Alice and Bob be the holders of the two files; Alice sent the file to Bob, and they wish to confirm it has been received without error. If Alice computes the hash of her file and sends it to Bob, and Bob computes the hash of his file, using the same M-bit hash function, and the two hashes match, then Bob can deduce that the two files are almost surely the same.

Example 12.6. What is the probability of a false negative, i.e., the probability, given that the two files do differ, that the two hashes are nevertheless identical?

If we assume that the hash function is random and that the process that causes the files to differ knows nothing about the hash function, then the probability of a false negative is 2^{-M}. □

A 32-bit hash gives a probability of false negative of about 10^{-10}. It is common practice to use a linear hash function called a 32-bit cyclic redundancy check to detect errors in files. (A cyclic redundancy check is a set of 32 parity-check bits similar to the 3 parity-check bits of the $(7,4)$ Hamming code.)

> To have a false-negative rate smaller than one in a billion, $M = 32$ bits is plenty, if the errors are produced by noise.

▷ Exercise 12.7.[*2*, p.203] Such a simple parity-check code only detects errors; it doesn't help correct them. Since error-*correcting* codes exist, why not use one of them to get some error-correcting capability too?

Tamper detection

What if the differences between the two files are not simply 'noise', but are introduced by an adversary, a clever *forger* called Fiona, who modifies the original file to make a forgery that purports to be Alice's file? How can Alice make a digital signature for the file so that Bob can confirm that no-one has tampered with the file? And how can we prevent Fiona from listening in on Alice's signature and attaching it to other files?

Let's assume that Alice computes a hash function for the file and sends it securely to Bob. If Alice computes a simple hash function for the file like the linear cyclic redundancy check, and Fiona knows that this is the method of verifying the file's integrity, Fiona can make her chosen modifications to the file and then easily identify (by linear algebra) a further 32-or-so single bits that, when flipped, restore the hash function of the file to its original value. *Linear hash functions give no security against forgers.*

We must therefore require that the hash function be *hard to invert* so that no-one can construct a tampering that leaves the hash function unaffected. We would still like the hash function to be easy to compute, however, so that Bob doesn't have to do hours of work to verify every file he received. Such a hash function – easy to compute, but hard to invert – is called a *one-way* *hash function.* Finding such functions is one of the active research areas of cryptography.

A hash function that is widely used in the free software community to confirm that two files do not differ is `MD5`, which produces a 128-bit hash. The details of how it works are quite complicated, involving convoluted exclusive-or-ing and if-ing and and-ing.[1]

Even with a good one-way hash function, the digital signatures described above are still vulnerable to attack, if Fiona has access to the hash function. Fiona could take the tampered file and hunt for a further tiny modification to it such that its hash matches the original hash of Alice's file. This would take some time – on average, about 2^{32} attempts, if the hash function has 32 bits – but eventually Fiona would find a tampered file that matches the given hash. To be secure against forgery, digital signatures must either have enough bits for such a random search to take too long, or the hash function itself must be kept secret.

> Fiona has to hash 2^M files to cheat. 2^{32} file modifications is not very many, so a 32-bit hash function is not large enough for forgery prevention.

Another person who might have a motivation for forgery is Alice herself. For example, she might be making a bet on the outcome of a race, without wishing to broadcast her prediction publicly; a method for placing bets would be for her to send to Bob the bookie the hash of her bet. Later on, she could send Bob the details of her bet. Everyone can confirm that her bet is consistent with the previously publicized hash. [This method of secret publication was used by Isaac Newton and Robert Hooke when they wished to establish priority for scientific ideas without revealing them. Hooke's hash function was alphabetization as illustrated by the conversion of *UT TENSIO, SIC VIS* into the anagram `CEIIINOSSSTTUV`.] Such a protocol relies on the assumption that Alice cannot change her bet after the event without the hash coming out wrong. How big a hash function do we need to use to ensure that Alice cannot cheat? The answer is different from the size of the hash we needed in order to defeat Fiona above, because Alice is the author of *both* files. Alice could cheat by searching for two files that have identical hashes to each other. For example, if she'd like to cheat by placing two bets for the price of one, she could make a large number N_1 of versions of bet one (differing from each other in minor details only), and a large number N_2 of versions of bet two, and hash them all. If there's a collision between the hashes of two bets of different types, then she can submit the common hash and thus buy herself the option of placing either bet.

Example 12.8. If the hash has M bits, how big do N_1 and N_2 need to be for Alice to have a good chance of finding two different bets with the same hash?

This is a birthday problem like exercise 9.20 (p.156). If there are N_1 Montagues and N_2 Capulets at a party, and each is assigned a 'birthday' of M bits, the expected number of collisions between a Montague and a Capulet is

$$N_1 N_2 2^{-M}, \tag{12.3}$$

[1] `http://www.freesoft.org/CIE/RFC/1321/3.htm`

so to minimize the number of files hashed, $N_1 + N_2$, Alice should make N_1 and N_2 equal, and will need to hash about $2^{M/2}$ files until she finds two that match. □

> Alice has to hash $2^{M/2}$ files to cheat. [This is the square root of the number of hashes Fiona had to make.]

If Alice has the use of $C = 10^6$ computers for $T = 10$ years, each computer taking $t = 1$ ns to evaluate a hash, the bet-communication system is secure against Alice's dishonesty only if $M \gg 2 \log_2 CT/t \simeq 160$ bits.

Further reading

The Bible for hash codes is volume 3 of Knuth (1968). I highly recommend the story of Doug McIlroy's `spell` program, as told in section 13.8 of *Programming Pearls* (Bentley, 2000). This astonishing piece of software makes use of a 64-kilobyte data structure to store the spellings of all the words of 75 000-word dictionary.

▶ 12.6 Further exercises

Exercise 12.9.[*1*] What is the shortest the address on a typical international letter could be, if it is to get to a unique human recipient? (Assume the permitted characters are `[A-Z,0-9]`.) How long are typical email addresses?

Exercise 12.10.[*2*, p.203] How long does a piece of text need to be for you to be pretty sure that no human has written that string of characters before? How many notes are there in a new melody that has not been composed before?

▷ Exercise 12.11.[*3*, p.204] Pattern recognition by molecules.

Some proteins produced in a cell have a regulatory role. A regulatory protein controls the transcription of specific genes in the genome. This control often involves the protein's binding to a particular DNA sequence in the vicinity of the regulated gene. The presence of the bound protein either promotes or inhibits transcription of the gene.

(a) Use information-theoretic arguments to obtain a lower bound on the size of a typical protein that acts as a regulator specific to one gene in the whole human genome. Assume that the genome is a sequence of 3×10^9 nucleotides drawn from a four letter alphabet $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$; a protein is a sequence of amino acids drawn from a twenty letter alphabet. [Hint: establish how long the recognized DNA sequence has to be in order for that sequence to be unique to the vicinity of one gene, treating the rest of the genome as a random sequence. Then discuss how big the protein must be to recognize a sequence of that length uniquely.]

(b) Some of the sequences recognized by DNA-binding regulatory proteins consist of a subsequence that is repeated twice or more, for example the sequence

$$\underline{\texttt{GCCCCC}}\texttt{CACCCCT}\underline{\texttt{GCCCCC}} \tag{12.4}$$

is a binding site found upstream of the alpha-actin gene in humans. Does the fact that some binding sites consist of a repeated subsequence influence your answer to part (a)?

▶ 12.7 Solutions

Solution to exercise 12.1 (p.194). First imagine comparing the string $\mathbf{x}$ with another random string $\mathbf{x}^{(s)}$. The probability that the first bits of the two strings match is 1/2. The probability that the second bits match is 1/2. Assuming we stop comparing once we hit the first mismatch, the expected number of matches is 1, so the expected number of comparisons is 2 (exercise 2.34, p.38).

Assuming the correct string is located at random in the raw list, we will have to compare with an average of $S/2$ strings before we find it, which costs $2S/2$ binary comparisons; and comparing the correct strings takes N binary comparisons, giving a total expectation of $S + N$ binary comparisons, if the strings are chosen at random.

In the worst case (which may indeed happen in practice), the other strings are very similar to the search key, so that a lengthy sequence of comparisons is needed to find each mismatch. The worst case is when the correct string is last in the list, and all the other strings differ in the last bit only, giving a requirement of SN binary comparisons.

Solution to exercise 12.2 (p.197). The likelihood ratio for the two hypotheses, $\mathcal{H}_0$: $\mathbf{x}^{(s)} = \mathbf{x}$, and $\mathcal{H}_1$: $\mathbf{x}^{(s)} \neq \mathbf{x}$, contributed by the datum 'the first bits of $\mathbf{x}^{(s)}$ and $\mathbf{x}$ are equal' is

$$\frac{P(\text{Datum} \,|\, \mathcal{H}_0)}{P(\text{Datum} \,|\, \mathcal{H}_1)} = \frac{1}{1/2} = 2. \tag{12.5}$$

If the first r bits all match, the likelihood ratio is 2^r to one. On finding that 30 bits match, the odds are a billion to one in favour of $\mathcal{H}_0$, assuming we start from even odds. [For a complete answer, we should compute the evidence given by the prior information that the hash entry s has been found in the table at $\mathbf{h}(\mathbf{x})$. This fact gives further evidence in favour of $\mathcal{H}_0$.]

Solution to exercise 12.3 (p.198). Let the hash function have an output alphabet of size $T = 2^M$. If M were equal to $\log_2 S$ then we would have exactly enough bits for each entry to have its own unique hash. The probability that one particular pair of entries collide under a random hash function is $1/T$. The number of pairs is $S(S-1)/2$. So the expected number of collisions between pairs is exactly

$$S(S-1)/(2T). \tag{12.6}$$

If we would like this to be smaller than 1, then we need $T > S(S-1)/2$ so

$$M > 2 \log_2 S. \tag{12.7}$$

We need *twice as many* bits as the number of bits, $\log_2 S$, that would be sufficient to give each entry a unique name.

If we are happy to have occasional collisions, involving a fraction f of the names S, then we need $T > S/f$ (since the probability that one particular name is collided-with is $f \simeq S/T$) so

$$M > \log_2 S + \log_2[1/f], \tag{12.8}$$

which means for $f \simeq 0.01$ that we need an extra 7 bits above $\log_2 S$.

The important point to note is the scaling of T with S in the two cases (12.7, 12.8). If we want the hash function to be collision-free, then we must have T greater than $\sim S^2$. If we are happy to have a small frequency of collisions, then T needs to be of order S only.

Solution to exercise 12.5 (p.198). The posterior probability ratio for the two hypotheses, $\mathcal{H}_+$ = 'calculation correct' and $\mathcal{H}_-$ = 'calculation incorrect' is the product of the prior probability ratio $P(\mathcal{H}_+)/P(\mathcal{H}_-)$ and the likelihood ratio, $P(\text{match} \mid \mathcal{H}_+)/P(\text{match} \mid \mathcal{H}_-)$. This second factor is the answer to the question. The numerator $P(\text{match} \mid \mathcal{H}_+)$ is equal to 1. The denominator's value depends on our model of errors. If we know that the human calculator is prone to errors involving multiplication of the answer by 10, or to transposition of adjacent digits, neither of which affects the hash value, then $P(\text{match} \mid \mathcal{H}_-)$ could be equal to 1 also, so that the correct match gives no evidence in favour of $\mathcal{H}_+$. But if we assume that errors are 'random from the point of view of the hash function' then the probability of a false positive is $P(\text{match} \mid \mathcal{H}_-) = 1/9$, and the correct match gives evidence 9:1 in favour of $\mathcal{H}_+$.

Solution to exercise 12.7 (p.199). If you add a tiny $M = 32$ extra bits of hash to a huge N-bit file you get pretty good error detection – the probability that an error is undetected is 2^{-M}, less than one in a billion. To do error *correction* requires far more check bits, the number depending on the expected types of corruption, and on the file size. For example, if just eight random bits in a megabyte file are corrupted, it would take about $\log_2 \binom{2^{23}}{8} \simeq 23 \times 8 \simeq 180$ bits to specify which are the corrupted bits, and the number of parity-check bits used by a successful error-correcting code would have to be at least this number, by the counting argument of exercise 1.10 (solution, p.20).

Solution to exercise 12.10 (p.201). We want to know the length L of a string such that it is very improbable that that string matches any part of the entire writings of humanity. Let's estimate that these writings total about one book for each person living, and that each book contains two million characters (200 pages with 10 000 characters per page) – that's 10^{16} characters, drawn from an alphabet of, say, 37 characters.

The probability that a randomly chosen string of length L matches at one point in the collected works of humanity is $1/37^L$. So the expected number of matches is $10^{16}/37^L$, which is vanishingly small if $L \geq 16/\log_{10} 37 \simeq 10$. Because of the redundancy and repetition of humanity's writings, it is possible that $L \simeq 10$ is an overestimate.

So, if you want to write something unique, sit down and compose a string of ten characters. But don't write `gidnebinzz`, because I already thought of that string.

As for a new melody, if we focus on the sequence of notes, ignoring duration and stress, and allow leaps of up to an octave at each note, then the number of choices per note is 23. The pitch of the first note is arbitrary. The number of melodies of length r notes in this rather ugly ensemble of Schönbergian tunes is 23^{r-1}; for example, there are 250 000 of length $r = 5$. Restricting the permitted intervals will reduce this figure; including duration and stress will increase it again. [If we restrict the permitted intervals to repetitions and tones or semitones, the reduction is particularly severe; is this why the melody of 'Ode to Joy' sounds so boring?] The number of recorded compositions is probably less than a million. If you learn 100 new melodies per week for every week of your life then you will have learned 250 000 melodies at age 50. Based on empirical experience of playing the game '`guess that tune`', it seems to me that whereas many four-note sequences are shared in common between melodies, the number of collisions between five-note sequences is rather smaller – most famous five-note sequences are unique.

In `guess that tune`, one player chooses a melody, and sings a gradually-increasing number of its notes, while the other participants try to guess the whole melody.

The *Parsons code* is a related hash function for melodies: each pair of consecutive notes is coded as `U` ('up') if the second note is higher than the first, `R` ('repeat') if the pitches are equal, and `D` ('down') otherwise. You can find out how well this hash function works at `http://musipedia.org/`.

Solution to exercise 12.11 (p.201). (a) Let the DNA-binding protein recognize a sequence of length L nucleotides. That is, it binds preferentially to that DNA sequence, and not to any other pieces of DNA in the whole genome. (In reality, the recognized sequence may contain some wildcard characters, e.g., the `*` in `TATAA*A`, which denotes 'any of `A`, `C`, `G` and `T`'; so, to be precise, we are assuming that the recognized sequence contains L non-wildcard characters.)

Assuming the rest of the genome is 'random', i.e., that the sequence consists of random nucleotides `A`, `C`, `G` and `T` with equal probability – which is obviously untrue, but it shouldn't make too much difference to our calculation – the chance that there is no other occurrence of the target sequence in the whole genome, of length N nucleotides, is roughly

$$(1-(1/4)^L)^N \simeq \exp(-N(1/4)^L), \tag{12.9}$$

which is close to one only if

$$N4^{-L} \ll 1, \tag{12.10}$$

that is,

$$L > \log N/\log 4. \tag{12.11}$$

Using $N = 3 \times 10^9$, we require the recognized sequence to be longer than $L_{\rm min} = 16$ nucleotides.

What size of protein does this imply?

- A weak lower bound can be obtained by assuming that the information content of the protein sequence itself is greater than the information content of the nucleotide sequence the protein prefers to bind to (which we have argued above must be at least 32 bits). This gives a minimum protein length of $32/\log_2(20) \simeq 7$ amino acids.
- Thinking realistically, the recognition of the DNA sequence by the protein presumably involves the protein coming into contact with all sixteen nucleotides in the target sequence. If the protein is a monomer, it must be big enough that it can simultaneously make contact with sixteen nucleotides of DNA. One helical turn of DNA containing ten nucleotides has a length of 3.4 nm, so a contiguous sequence of sixteen nucleotides has a length of 5.4 nm. The diameter of the protein must therefore be about 5.4 nm or greater. Egg-white lysozyme is a small globular protein with a length of 129 amino acids and a diameter of about 4 nm. Assuming that volume is proportional to sequence length and that volume scales as the cube of the diameter, a protein of diameter 5.4 nm must have a sequence of length $2.5 \times 129 \simeq 324$ amino acids.

(b) If, however, a target sequence consists of a twice-repeated sub-sequence, we can get by with a much smaller protein that recognizes only the sub-sequence, and that binds to the DNA strongly only if it can form a *dimer*, both halves of which are bound to the recognized sequence. Halving the diameter of the protein, we now only need a protein whose length is greater than $324/8 = 40$ amino acids. A protein of length smaller than this cannot by itself serve as a regulatory protein specific to one gene, because it's simply too small to be able to make a sufficiently specific match – its available surface does not have enough information content.