# AQA Computer Science GCSE

## 3.3 Fundamentals of data representation

Advanced Notes

# 3.3.1 Number bases

## What are number bases?

A number base (or numeral system) determines how values are represented using digits. The most common bases are:

- Decimal (Base 10) - used by humans for counting
- Binary (Base 2) - used by computers to represent all data and instructions
- Hexadecimal (Base 16) - used by programmers for compact representation

## Decimal (base 10)

Decimal is the number base that humans use to count, perhaps because we have ten fingers. Decimal uses the ten digits 0 through to 9 to represent numbers.

Each digit in a decimal number has a place value based on powers of 10. The value of a digit depends on its position within the number. This is illustrated by the table below, which shows how the decimal number 237 is constructed using place values.

| $10^2$ | $10^1$ | $10^0$ |
|---|---|---|
| 100 | 10 | 1 |
| 2 | 3 | 7 |

`237 = (2×100) + (3×10) + (7×1)`

## Binary (base 2)

Binary is used by computer systems to store and represent all data and instructions. This is because it has only two states, 0 or 1.

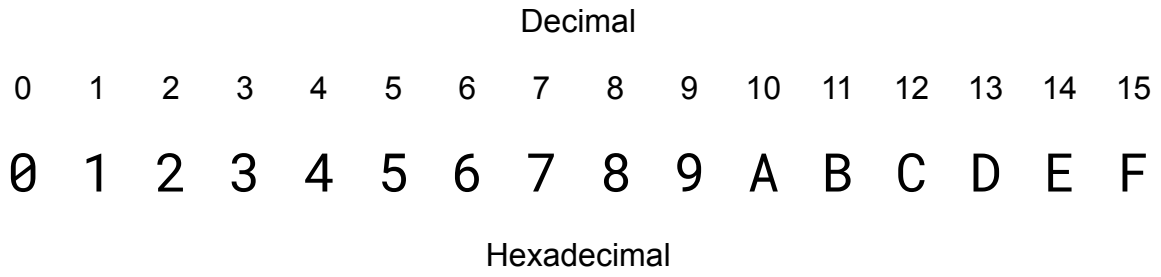
Each digit in a binary number has a place value based on powers of 2. This is illustrated by the table below, which shows how the binary number 1011 is constructed using place values - making it equal to 11 in decimal.

| $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|
| 8 | 4 | 2 | 1 |
| 1 | 0 | 1 | 1 |

`1011 = (1×8) + (0×4) + (1×2) + (1×1) = 11 (decimal)`

## Hexadecimal (base 16)

In contrast to decimal, hexadecimal uses the digits 0 through to 9 followed by the uppercase characters A to F to represent the decimal numbers 0 to 15.




Of all the number bases covered by this course, hexadecimal is the most compact. This means that it can represent the same number as binary or decimal while using far fewer digits. Each character in hexadecimal represents four bits in binary.

Each digit in a decimal number has a place value based on powers of 16. This is illustrated by the table below, which shows how the hexadecimal value 2F is constructed using place values - making it equal to 47 in decimal.

| $16^1$ | $16^0$ |
|---|---|
| 16 | 1 |
| 2 | 15 (because F represents 15) |

```
2F = (2×16) + (15×1)  = 47 (decimal)
```

### Why use hexadecimal?

Hexadecimal is easier for people to read than binary, and it takes less time to type than binary. Therefore, hexadecimal representation is used because it is easier for humans to read and work with. However, hexadecimal does not offer any advantage to computers: computers always represent numbers using binary.

### Bit patterns represent data

A binary value could represent:

- A **number** (e.g. `00000101` = `5`)
- A **character** (e.g. ASCII code for `A`)
- A **pixel** in an image
- A **sample** in audio

The datatype a bit pattern represents depends on how it is interpreted by the program.

# 3.3.2 Converting between number bases

## What is base conversion?

Base conversion is the process of changing a number from one number system to another - specifically, between:

- Binary (base 2)
- Decimal (base 10)
- Hexadecimal (base 16)

You must be able to work with whole numbers only, up to a maximum value of:

- Decimal: 255
- Binary: 11111111 (8 bits)
- Hex: FF

## Converting Decimal ↔ Binary

**To convert binary → decimal:**

You can convert from binary to decimal by using place value headers. Starting with one and increasing in powers of two, placing larger values to the left of smaller values. For example, the binary number 10110010 could have place value headers added as follows:

| 128 ($2^7$) | 64 ($2^6$) | 32 ($2^5$) | 16 ($2^4$) | 8 ($2^3$) | 4 ($2^2$) | 2 ($2^1$) | 1 ($2^0$) |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |

The binary number could then be converted to decimal by adding together all of the place values with a binary one below them.

$$128 + 32 + 16 + 2 = 178$$

So the binary number 10110010 is equivalent to the decimal number 178.

**To convert decimal → binary:**

When converting from decimal to binary, you use the same place value headers. Starting from the left hand side, you place a one if the value is less than or equal to your number, and a zero otherwise.

Once you've placed a one, you must subtract the value of that position from your number and continue as before, until your decimal number becomes 0.

Let's say we're converting the number 53 to binary. First, write out your place value headers in powers of two. Keep going until you've written a value that is larger than your number. For 53, we're going to go up to 64.

| 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|

Now, starting from the left, compare the place value to your number. 64 is greater than 53 so we place a 0 under 64.

| 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|
| 0 | | | | | | |

Moving to the right, we see that 32 is lower than 53, so we place a 1 under 32.

| 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|
| 0 | 1 | | | | | |

Because we've placed a 1, we have to subtract 32 from 53 to find what's left to be represented. In this case, 53 - 32 = 21.

We move to the right again and find 16, which is lower than 21, so we place a 1 under 16.

| 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | | | | |

Again, because we've placed a 1, we have to calculate a new value. 21 - 16 = 5.

Moving right, we find 8. This is larger than 5 so we place a 0.

| 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | | | |

After moving right again, we find 4. As 4 is lower than 5, we place a 1.

| 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 | | |

Having placed a 1, we must again calculate a new value. 5 - 4 = 1.

Moving right to find 2, we place a 0 as 2 is greater than 1.

| 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 | 0 | |

Moving right for the last time, we have 1. 1 = 1 so we place a 1.

| 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 | 0 | 1 |

Now that we've placed a 0 or a 1 under each place value, we have our answer. Although it's acceptable to remove any leading 0s, it may be preferable to add 0s to the start of your answer to make it a whole number of bytes (a multiple of 8 bits).

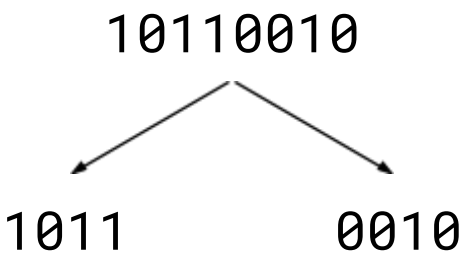
$$53 = 0110101 = 110101 = 00110101$$

## Converting Binary ↔ Hexadecimal

**To convert binary → hex:**

In order to convert from binary to hexadecimal, the binary number must first be split into nibbles. A nibble is four binary bits, or half a byte.

For example, the binary number 10110010 would be split into two nibbles:




Each binary nibble is then converted to decimal as in the previous example:

| 8 | 4 | 2 | 1 |
|---|---|---|---|
| 1 | 0 | 1 | 1 |

8 + 2 + 1 = 11

| 8 | 4 | 2 | 1 |
|---|---|---|---|
| 0 | 0 | 1 | 0 |

2 = 2

Once each nibble has been converted to decimal, the decimal value can be converted to its hexadecimal equivalent like so:

11 = B

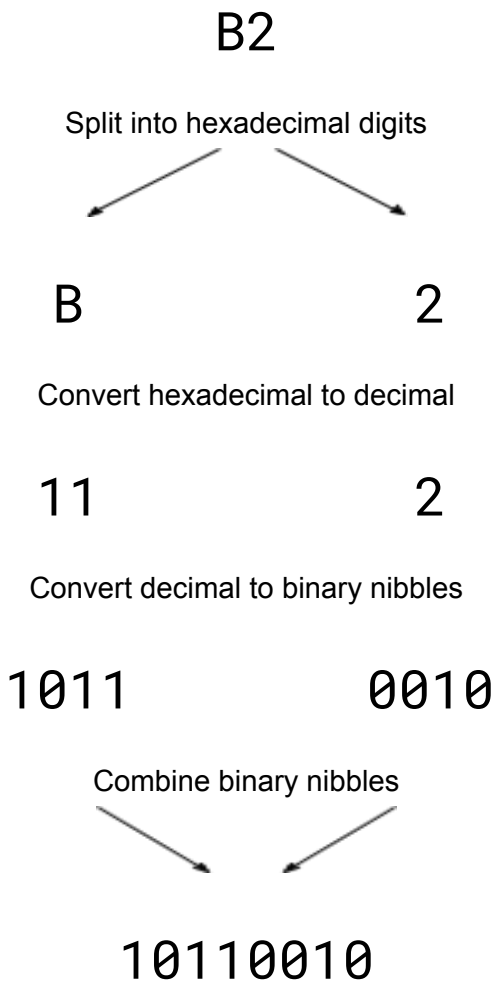
2 = 2

Finally, the hexadecimal digits are concatenated to form a hexadecimal representation:

10110010 = B2

**To convert hex → binary**:

First, convert each hexadecimal digit to a decimal digit and then to a binary nibble before combining the nibbles to form a single binary number.




## Converting Decimal ↔ Hexadecimal

**To convert decimal → hex:**

Combining the steps above:

1. Begin by converting the decimal number into binary
2. Convert this binary number to hexadecimal

**To convert hex → decimal:**

Combining the steps above:

1. Begin by converting the hexadecimal number into binary
2. Convert this binary number to decimal.

# 3.3.3 Units of information

The fundamental (smallest) unit of information is the bit (binary digit), and larger units are made up of bits grouped together. A collection of 8 bits is called a byte. A bit is notated with a lowercase b whereas a byte uses the uppercase B.

You'll often come across the following prefixes used for decimal numbers.

| Unit | Symbol | Relative size |
|---|---|---|
| Bit | b | 1 bit |
| Byte | B | 8 bits |
| Kilobyte | kB | 1,000 bytes |
| Megabyte | MB | 1,000 kilobytes |
| Gigabyte | GB | 1,000 megabytes |
| Terabyte | TB | 1,000 gigabytes |

***Note about prefixes:*** *the specification uses decimal (base-10) prefixes, as shown in the table above. These differ from binary prefixes (e.g. kibibyte = 1,024 bytes), but you only need to know about the prefixes in the table above for the AQA GCSE Computer Science (8525) exam.*

## Comparing quantities of bits

You should be able to compare quantities of bits using the prefixes shown in the table above.

**For example:**

- 1 GB = 1,000,000,000 bytes
- 1 MB = 1,000,000 bytes
  So: 1 GB = 1,000 MB
  1 MB = 1,000 kB

# 3.3.4 Binary arithmetic

Binary arithmetic involves performing mathematical operations using binary numbers (0s and 1s).

## Binary addition

When adding binary numbers, there are four important rules to remember:

| Binary add | Result | Carry |
|---|---|---|
| 0 + 0 | 0 | 0 |
| 1 + 0 | 1 | 0 |
| 1 + 1 | 0 | 1 (carry) |
| 1 + 1 + 1 | 1 | 1 (carry) |

You'll only be expected to add up to three binary numbers of up to 8 bits.

**Example**

Add binary integers 1011 and 1110.

| | | | | |
|---|---|---|---|---|
| | 1 | 0 | 1 | 1 |
| + | 1 | 1 | 1 | 0 |

Place the two binary numbers above each other so that the digits line up.

| | | | | |
|---|---|---|---|---|
| | 1 | 0 | 1 | 1 |
| + | 1 | 1 | 1 | 0 |
| | | | | 1 |

Starting from the least significant bits (the right hand side), add the values in each column and place the total below. For the first column (highlighted), rule 2 from above applies.

| | | | | |
|---|---|---|---|---|
| | 1 | 0 | 1 | 1 |
| + | 1 | 1 | 1 | 0 |
| | | $^{1}$ | 0 | 1 |

Move on to the next column. This time rule 3 applies. In this case there is a carry digit. Place a 1 in small writing under the next most significant bit's column.

| | | | | |
|---|---|---|---|---|
| | 1 | 0 | 1 | 1 |
| + | 1 | 1 | 1 | 0 |
| | $^{1}$ | 0$^{1}$ | 0 | 1 |

On to the next column, where there is a 0, a 1 and a small 1. In this case, rule 3 applies again. Therefore the result is 10. Because 10 is two digits long, the 1 is written in small writing under the next most significant bit's column.

| | | | | |
|---|---|---|---|---|
| | 1 | 0 | 1 | 1 |
| + | 1 | 1 | 1 | 0 |
| 1 | 1$^{1}$ | 0$^{1}$ | 0 | 1 |

Moving on to the most significant column where there are three 1s. Rule 4 applies, so the result for this column is 11. The first digit of the result is written under the next most significant bit's column, but it can be written full size as there are no more columns to add.

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 |

Finally, the result is read off from the full size numbers at the bottom of each column. In this case, 1011 + 1110 = 11001.

## Binary shifts

A binary shift involves moving the bits of a binary number left or right. Bits shifted from the end of the register are lost and zeros are shifted in at the opposite end of the register.

There are two types of binary shift:

- Left shift → moves all bits to the left (adds 0s on the right)
    - Same as multiplying by 2 for each place shifted
- Right shift → moves all bits to the right (adds 0s on the left)
    - Same as dividing by 2 for each place shifted

### Example

In this example, we'll apply a binary left shift of 1 to the original binary number `00101100`. The effect of this is to multiply 44 by 2, making 88.

| | |
|---|---|
| Original: | `00101100 (44)` |
| Shifted: | `01011000 (88)` |

## Why use binary shifts?

- To multiply or divide by powers of 2
- Used in low-level graphics, bitmasking, compression, and encryption

# 3.3.5 Character encoding

## What is character encoding?

Character encoding is the process of converting characters (letters, numbers, symbols) into binary so that they can be stored and processed by a computer's hardware. This is necessary because computers can only store and process binary data.

A character set, such as ASCII or Unicode, is a collection of characters and their corresponding binary values. Every character is assigned a unique binary code, known as a character code, using a standard such as ASCII or Unicode. Character codes are grouped and they run in sequence. For example in ASCII 'A' is coded as 65, 'B' as 66, and so on, meaning that the codes for the other capital letters can be calculated once the code for 'A' is known. This pattern also applies to other groupings such as lower case letters and digits.

## ASCII (American Standard Code for Information Interchange)

- Uses 7 bits to represent each character
- Can store 128 ($2^7$) characters
- Includes:
    - English letters (uppercase & lowercase)
    - Digits 0–9
    - Common symbols (@, #, etc.)
    - Control codes (like newline)

## Unicode

- Uses 8-48 bits to represent each character, allowing it to represent a much wider range of different characters than ASCII
- Supports many different languages (not just the Latin alphabet but also alphabets like Arabic, Cyrillic, Greek and Hebrew), and more symbols (such as emojis). This means that data such as text can be represented in a wider range of languages, making computers more accessible worldwide.
- Unicode uses the same codes as ASCII up to 127.

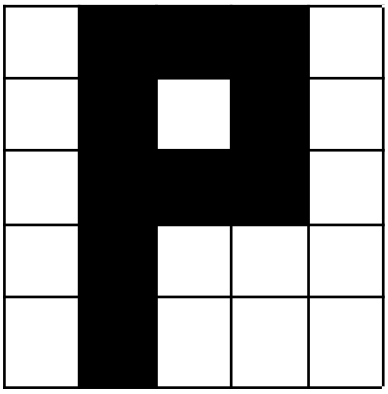
| Feature | ASCII | Unicode |
|---|---|---|
| Bit length | 7 bits (128 characters) | 8–32 bits (over 100k characters) |
| Language support | English only | Worldwide character support |
| Use today | Legacy systems | Standard in modern systems |

# 3.3.6 Representing images

**How are images represented in a computer?**

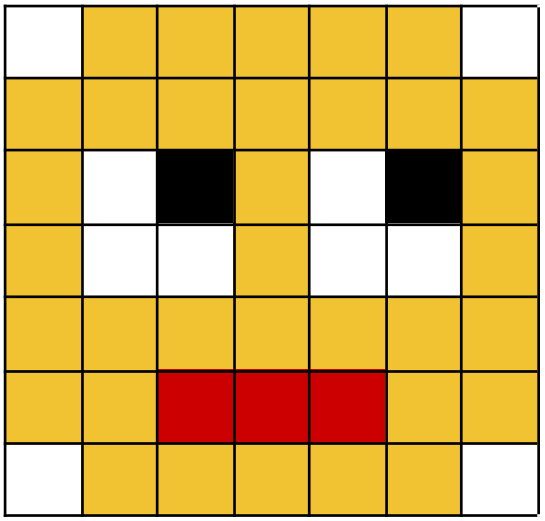
Digital images, known as bitmaps, are made up of tiny squares called pixels (short for "picture elements"). A pixel is a single point in an image. Each pixel has a colour value, and this is stored in binary.

The value assigned to a pixel determines the colour of the pixel. The example below shows the binary representation of a simple bitmap image in which a 1 represents a black pixel and a 0 represents a white pixel.



```
0 1 1 1 0
0 1 0 1 0
0 1 1 1 0
0 1 0 0 0
0 1 0 0 0
```

The number of bits assigned to each pixel in an image is called its colour depth. In the example above, each pixel has been assigned one bit, allowing for 2 ($2^1$) different colours to be represented. If a colour depth of two bits were used, there would be four ($2^2$) different colours that each pixel could take, represented by the bit patterns 00, 01, 10 and 11.



```
00 11 11 11 11 11 00
11 11 11 11 11 11 11
11 00 01 11 00 01 11
11 00 00 11 00 00 11
11 11 11 11 11 11 11
11 11 10 10 10 11 11
00 11 11 11 11 11 00
```

## Calculating image size and file size

$$Image\ size\ =\ width \times height$$

$$File\ size\ =\ width \times height \times colour\ depth$$

(where file size is to be calculated in bits, width and height are measured in pixels, and colour depth is measured in bits)

In order to calculate the storage required to represent a bitmap image in bits, multiply the image size by the bit depth. To calculate the file size of an image in bytes, divide the file size in bits by 8.

### Example

The picture of the face has 7 × 7 = 49 pixels, each of which has a two bit colour-depth, so it requires 98 bits to be represented.

$$File\ size\ =\ 7 \times 7 \times 2\ =\ 98\ bits$$

## Effect on image size, quality and file size

Looking at the equation above, higher widths, higher heights and higher colour depths will all increase an image's file size.

| Increases in… | Effect on… |
|---|---|
| Width | Larger image size & higher file size |
| Height | Larger image size & higher file size |
| Colour depth | Higher quality & higher file size |

# 3.3.7 Representing Sound

## How is sound represented in a computer?

Analogue signal



Digital signal

Sound is analogue, meaning that its signal is a continuous wave that can take any value, not having a singular value. Computers cannot store continuous sound waves, so they take regular snapshots (samples) of the sound wave's amplitude. A sample is a measure of amplitude at a point in time - each sample is stored as a binary number.

The sampling rate is the number of samples taken in a second and is usually measured in hertz (1 hertz = 1 sample per second).

The sample resolution is the number of bits per sample.

### Calculating sound file size

$$File\ size\ =\ sampling\ rate \times sample\ resolution \times duration$$

(where file size is to be calculated in bits, sampling rate is measured in Hz, sample resolution is measured in bits and duration is measured in seconds)

To calculate the file size of a sound file in bytes, divide the file size in bits by 8.

**Example**

For a sound file with a:

- Sample rate = 44,100 Hz
- Sample resolution = 16 bits
- Duration = 10 seconds

$$File\ size\ =\ 44100 \times 16 \times 10\ =\ 7056000\ bits$$

## Effect on quality and file size

| Increases in… | Effect on… |
|---|---|
| Sampling rate | Higher sound quality & larger file size |
| Bit depth | More accurate samples & larger file size |

# 3.3.8 Data compression

## What is data compression?

Data compression is the process of reducing the file size of digital data without losing the original information (or with minimal acceptable loss). It is used to save storage space and speed up transmission.

## Why compress data?

- Saves storage space
- Speeds up file transfer
- Reduces bandwidth usage
- Helps with faster downloads and streaming

## Types of compression

### Lossy compression

When using lossy compression, some information is lost in the process of reducing the file's size, which can never be fully restored to the original. This could cause the quality of the file to be slightly reduced.

**Used for:**

- Images
- Audio
- Video

| Pros | Cons |
|---|---|
| ✔ Smaller file size | ✘ Loss of quality |
| ✔ Faster to send/store | ✘ Irreversible (original data gone) |

**Lossless compression**

In contrast to lossy compression, there is no loss of information when using lossless compression. The size of a file can be reduced without decreasing its quality. Lossless compression methods use algorithms to find and compress patterns (e.g. repeated data).

Two methods of lossless compression include Huffman coding and run length encoding (RLE).

**Used for:**

- Text files
- Code
- Images
- Audio
- Video
- ZIP archives

| Pros | Cons |
|---|---|
| ✔ No loss of quality | ✘ Less reduction in size compared to lossy |
| ✔ Reversible | |

## Huffman coding

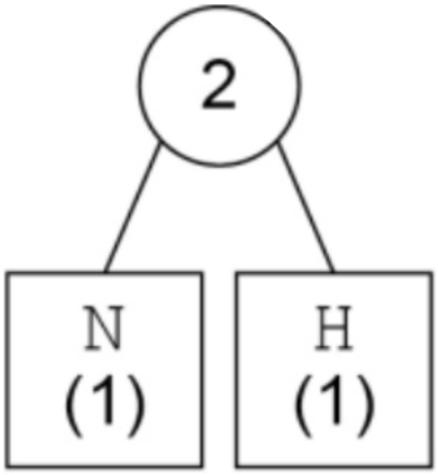
Using the ASCII character encoding, every character is encoded using the same number of bits. However, it is possible to use a smaller number of bits for some characters. Huffman coding reduces the size of a file by using fewer bits for frequently occurring characters and more bits for rare ones.

Let's look at how to create a Huffman tree using an example string, COMPUTER SCIENCE IS THE BEST SUBJECT.

First, calculate how many times each character appears in the string, and rank these from most to least frequent. 'SP' represents space.

| Character | Frequency |
|---|---|
| E | 6 |
| SP | 4 |
| S | 4 |
| C | 4 |
| T | 4 |
| U | 2 |
| B | 2 |
| I | 2 |
| J | 1 |
| H | 1 |
| N | 1 |
| P | 1 |
| R | 1 |
| O | 1 |
| M | 1 |

Next, select the two characters from the bottom of the ranking and place them, along with their frequency, into a new binary tree. The new node's frequency = the sum of the two frequencies. For example, for N and H, the node would look like this:

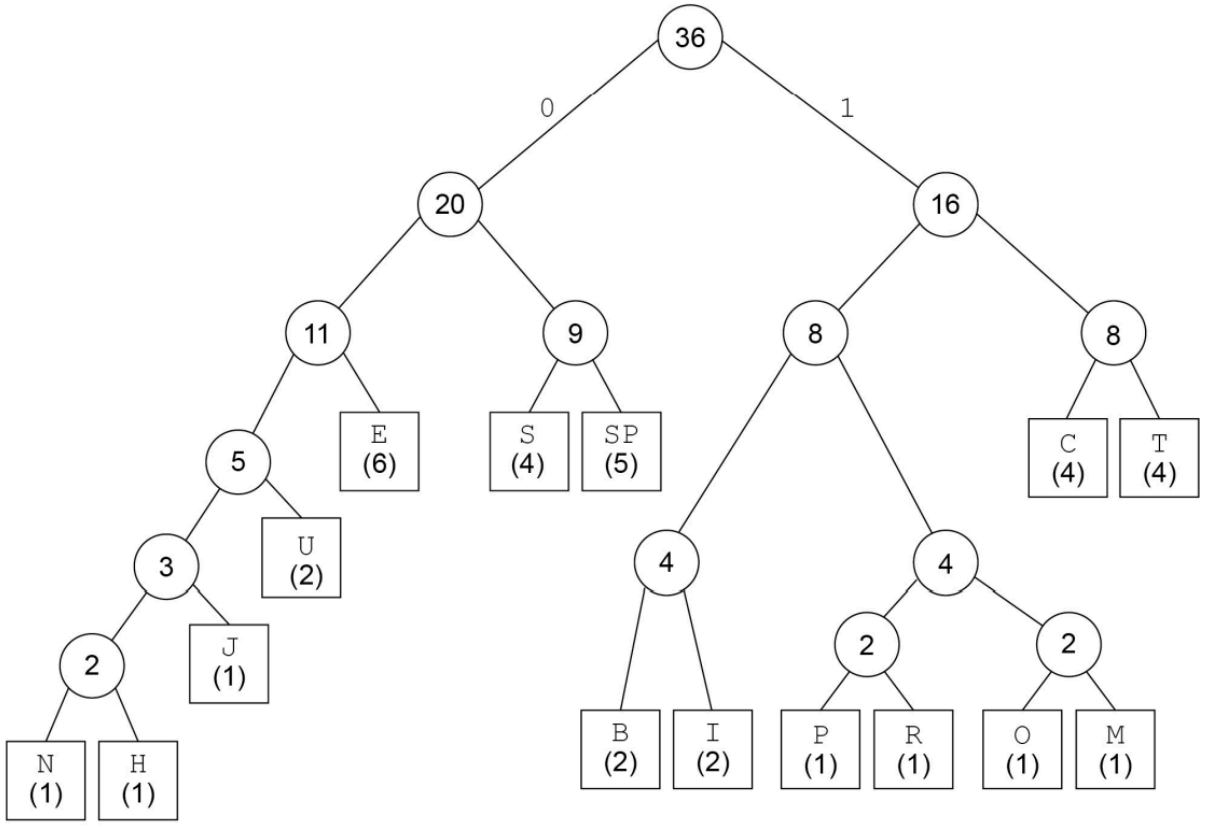



Then update the table showing the combined characters.

| Character | Occurrences |
|---|---|
| E | 6 |
| SP | 4 |
| S | 4 |
| C | 4 |
| T | 4 |
| U | 2 |
| B | 2 |
| I | 2 |
| **NH** | **2** |
| J | 1 |
| H | 1 |
| N | 1 |
| P | 1 |
| R | 1 |

Take the next two least frequent characters and use them to create a node, and repeat this process until all the characters are combined.

Once you've done this, assign a 0 to the initial left-hand branch and a 1 to the initial right-hand branch. While you can label all left-hand branches with 0s and all right-hand branches with 1s, it's enough to just mark the first 0 and 1 as this process can be time-consuming for larger trees.




**(SP represents space)**

Now, each character has a unique bit pattern, using these 1s and 0s. For example, the bit pattern for S is 010, because to get to S from the root node you take the left node (0), followed by the right node (1), followed by the left node (0), giving 010.

Characters that are more frequent have a shorter bit pattern, making the representation as compact as possible.

**Calculating required bits in Huffman coding and ASCII**

We can carry out calculations to determine the number of bits saved by compressing a piece of data using Huffman coding compared with ASCII.

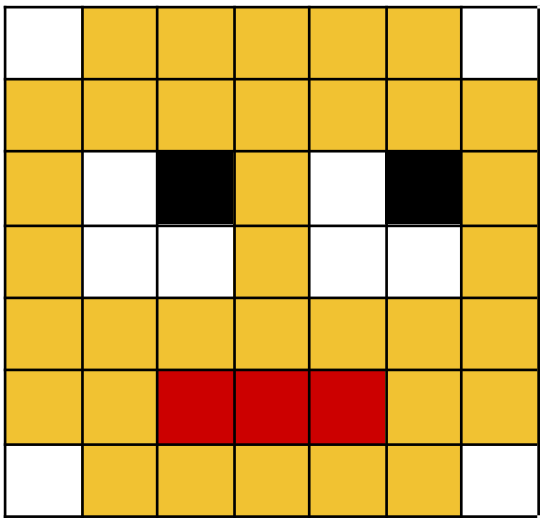
To calculate the number of bits required to represent a piece of data in ASCII, multiply the number of characters by 7.

To calculate the number of bits required to represent a piece of data using Huffman coding, Multiply the length of each character's Huffman code by its frequency. For example, if the character 'S' has a Huffman code of '010' (3 bits) and appears 100 times, it contributes 3 * 100 = 300 bits to the total. Sum these products for all unique characters in the data, to find the total number of bits needed to store the original data using Huffman coding.

To calculate the number of bits saved, subtract the number of bits in Huffman coding from the number of bits in ASCII.

### Run length encoding (RLE)

Run length encoding (RLE) reduces the size of a file by removing repeated information and replacing it with one occurrence of the repeated information followed by the number of times it is to be repeated.



$$00\ 11_5\ 00$$
$$11_7$$
$$11\ 00\ 01\ 11\ 00\ 01\ 11$$
$$11\ 00_2\ 11\ 00_2\ 11$$
$$11_7$$
$$11_2\ 10_3\ 11_2$$
$$00\ 11_5\ 00$$

The example uses the image of a face that was represented as a bitmap image earlier in these notes. Using RLE to replace repeated pixels with one pixel value and a number of repetitions has reduced the storage space required to represent the image.

The third row of pixels in the image has no repeated values and as such, couldn't be compressed by RLE. This highlights the fact that not all data is suitable for compression by run length encoding. For example, text is not suited to RLE at all, as it is unlikely to have many 'runs' of repeated letters.