O Poder do Byte.
(Por PaulOthar)

Disclamer: Eu posso ter cometido algum erro em algum momento, portanto não trate como verdade absoluta!

Você(Leitor) Já Deve Estar Careca de Saber que um Byte é composto por 8 Bits, e que certos Tipos primitivos das linguagens de programação são compostos por múltiplos Bytes (Como inteiro, que é composto por 4 Bytes).

Mas você deve ter se perguntado algum dia: "Idai porra? Foda-se a quantidade de Bytes que eu estou usando."

Bem... Considerando a evolução dos computadores, Sim, foda-se a quantidade que usamos em tempo de execução... PORÉM nem sempre podemos nos dar ao luxo de esbanjar espaço, Ou, certas operações dependem de uma velocidade de transferência superior, para se tornar eficiente.

É ai que o estudo dos Bytes entra, pois veja bem, caro leitor, nós somos mamacos, e mamacos são deveras criativos, e fruto da nossa criatividade vem das Abstrações,e sem as abstrações, a computação não seria oquê é hoje.

Vejamos um exemplo do porquê:

Um Byte é Composto por 8 Bits:

0000 0000

a primeira abstração que podemos pensar em 8 números binários, é a de formar números maiores, sendo cada casa decimal um número multiplo de 2, a ser somado com os demais:
(2**7)(2**6)(2**5)(2**4) (2**3)(2**2)(2**1)(2**0)

2**0 = 1
2**1 = 2
2**2 = 4
2**3 = 8
2**4 = 16
2**5 = 32
2**6 = 64
2**7 = 128

0000 0010 = 0+0+0+0 + 0+0+2+0 = 2
0000 1000 = 0+0+0+0 + 8+0+0+0 = 8
1010 1010 = 128+0+32+0 + 8+0+2+0 = 170
1111 1111 = 128+0+32+0 + 8+4+2+1 = 255

Note que não há número maior que 255, e não há número menor que 0.... portanto números negativos non eczistem, eles são criados apartir de nossa abstração.
A forma em que interpretamos números negativos, é com o último bit, sendo 0 para positivo, e 1 para negativo:

0000 0000 = 0
0000 1111 = 15
1000 1111 = -15
1111 1111 = -127

Como é uma abstração, a forma em que números negativos são tratados pode variar de linguágem para linguágem. Exemplo, em java, 1000 0000 é -128, e 1000 0001 é -127, portanto 1111 1111 é -1.

Matemática é só um rumo que podemos tomar com Bytes, o uso depende de nossa imaginação, como por exemplo, a de usar Bytes como endereço para um símbolo qualquer.
Técnica usada para escrever este texto, visto que Letras no computador são apenas símbolos com endereços de 1 Byte.

Exemplos:

0101 0000 = 080 = 'P'
0110 0001 = 097 = 'a'
0111 0101 = 117 = 'u'
0110 1100 = 108 = 'l'

Não existem limites para a interpretação de um Byte. Considere por exemplo uma loja de roupa qualquer, que confecciona camisas customizadas:

A loja oferece 4 opções de cada:
Tecido:["Seda","Lã","Malha","Veludo"]
Tamanho:["Olívia Palito","Médio","Grande","Aquilo é uma Lua?"]
Estampa:["Frô","Bob Esponja Fumando Crack","Abaporu","Auto retrato barroco do Lula molusco"]
Cor:["Vermelho","Verde","Azul","Preto"]

Uma pessoa qualquer registraria esses dados como:

{"Seda","Grande","Bob Esponja Fumando Crack","Preto"}

Porém, você só precisa usar 1 Byte para definir todas as opções de blusa:

(Tecido)(Tamanho) (Estampa)(Cor) = (0)(2) (1)(3) = (00)(10) (01)(11) = 0010 0111

O limite da abstração é sua imaginação, e para isso a maioria dos sistemas fornece operações especiais para você lidar com estas abstrações.
Operações em Byte são chamados de Bitwise:

And(E) : "&"

Se você sempre se perguntou o porquê que operações lógicas em linguágens sérias são representados por "&&" ao invés de "&", é por causa disso.

And em Operações Bitwise, são chamados de Máscara(Mask),e funcionam assim:

Byte:
0110 1001

Máscara:
1111 0000

0110 1001 & 1111 0000 = 0110 0000

Você vai ler apenas os uns que combinam uns com os outros.
No caso nossa mascara quer ler apenas os 4 ultimos bits,e no nosso Byte temos apenas o Penultimo e o Antepenúltimo Bit com 1, portanto apenas eles aparecerão.
todos os bits com 0 na máscara serão obrigatóriamente ignorados.

Or(Ou) : "|"

Não vejo muita utilidade para este, mas eis como funciona:

Byte:
0000 0011

Outro Byte:
0000 1101

0000 0011 | 0000 1101 = 0000 1111

Você vai basicamente sobrepor os bits, exibindo a junção dos dois Bytes, ignorando qualquer sobreposição de 1.

Xor(Ou Exclusivo) : "^"

Assim como o Ou, não vejo muita utilidade:

Byte:
1010 0101

Outro Byte:
1100 0011

1010 0101 ^ 1100 0011 = 0110 0110

Onde Houver a comparação de 1 com 0, e 0 com 1, vc põe 1, onde houver comparação de 0 com 0 e 1 com 1, vc põe 0.

A única Ultilidade que consigo pensar para o Xor é um inversor:

1001 0110
1111 1111

1001 0110 ^ 1111 1111 = 0110 1001

Bitshift : "<<",">>"

Esta operação faz o Byte mover as casas para esquerda ou direita, X vezes, porém também pode ser representado por (Byte*2**X) para mover para a esquerda, ou (Byte/2**X) para mover para a direita

Byte:
0000 1100

0000 1100 << 3 = 0110 0000

CUIDADO:
Informações podem ser perdidas se ultrapassarem os limites do Byte:

0000 1111 >> 1 = 0000 0111
0000 0111 << 1 = 0000 1110 (Informação perdida)

De volta ao Exemplo da loja de roupas, se eu quiser ler qual o tipo de tecido, basta eu fazer:

(XXXX XXXX >> 6) & 0000 0011

Porquê:

0100 0000 >> 6 = 0000 0001
0000 0001 & 0000 0011 = 0000 0001
0000 0001 = 1

Logo o tecido é o de número 1.

Atente-se ao fato que a maioria das linguagens suporta o 'formato' de representação em Binário:

0b11 = 0000 0011 = 3
0b00000011 = 0000 0011 = 3
0b000011 = 0000 0011 = 3

e em Hexadecimal:
0x03 = 0000 0011 = 3
0xf0 = 1111 0000 = 240

CUIDADO PARA NÃO COLOCAR ESPAÇO, POIS NÃO É VÁLIDO:
0b0000 0000 <= Dá Erro

Com este conhecimento você já pode evitar o desperdício de Bytes em operações do dia a dia.
Afinal, quem caralho precisa usar todos os 4 Bytes de um inteiro?

Porém, com o poder das abstrações podemos Espremer ainda mais desempenho, pois veja bem, um belo dia, um maluco chamado Huffman criou um algorítimo para comprimir Texto em Bits!

Lembrando que 1 Caractere Custa geralmente 1 Byte, ou 8 Bits.

Vamos analisar oquê ele fez:

Considere a Palavra: "abracadabra"

a palavra contém 5 "a", e cada um está ocupando 1 Byte.... praquê estamos compactuando com esta insanidade? não estamos usando a tabela ASCII inteira, só alguns poucos caracteres!

então vamos contar quantas vezes cada Letra aparece na palavra:

b = 2
d = 1
a = 5
r = 2
c = 1

Vamos ordenar em ordem decrescente:

a = 5
b = 2
r = 2
d = 1
c = 1

agora ordenado, vamos pegar os dois últimos e juntar, em formato de galho de árvore binária, e reinserilo de

volta na lista, com sua frequência somada:

a = 5
(c,d) = 2
b = 2
r = 2

No caso, se tiverem a mesma frequência, pouco importa a ordem, porém, se um for maior que o outro, o menor fica na esquerda, e o maior fica na direita.

Repetimos o processo até haver apenas 1 elemento na lista:

a = 5
(b,r) = 4
(c,d) = 2

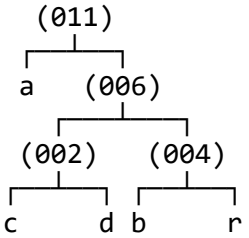
Neste caso, os dois ultimos são conjuntos, neste caso, iremos colocar o Menos frequente na esquerda, e o mais frequente na direita:

((c,d),(b,r)) = 6
a = 5

Ultima Iteração:

(a,((c,d),(b,r))) = 11

Vamos Montar esta árvore:



Vamos escrever o percurso da árvore para cada letra, sendo esquerda 0, e direita 1(escrita e leitura de esquerda para a direita):

a = 0
b = 110
r = 111
d = 101
c = 100

Agora vamos usar estes percursos como códigos, e substitui-los na palavra:

abracadabra

0 b r 0 c 0 d 0 b r 0

0 110 r 0 c 0 d 0 110 r 0

0 110 111 0 c 0 d 0 110 111 0

0 110 111 0 c 0 101 0 110 111 0

0 110 111 0 100 0 101 0 110 111 0

Código final:

01101110100010101101110

23 Bits! Se fosse-mos considerar cada caractere 8 Bits, teríamos o gasto de 88 Bits!
Ou seja, economizamos 65 Bits, graças ao algoritmo de Huffman!

Note que o caractere 'a' que é o mais frequente, tem o menor código, isso é uma das provas que este algoritmo pode optimizar significativamente o espaço de qualquer texto.

Mas lembre-se, o algoritmo de Huffman não é exclusivo para letras, você pode abstrai-lo e considerar um grupo qualquer, usando seus componentes para formar a Lista, depois árvore, depois código.

As possibilidades são ilimitadas, e a optimização é inevitável.