NJIT
New Jersey's Science & Technology University

A. V. GERBESSIOTIS

1 Powers of two

Definition 1.1 (Powers of 2). *The expression 2^n means the multiplication of n twos.*

Therefore, $2^2 = 2 \cdot 2$ is a 4, $2^8 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$ is 256, and $2^{10} = 1024$. Moreover, $2^1 = 2$ and $2^0 = 1$. Several times one might write $2**n$ or $2\hat{}n$ for 2^n (^ is the hat/caret symbol usually co-located with the numeric-6 keyboard key).

Power	Value
2^0	1
2^1	2
2^4	16
2^8	256
2^{10}	1024
2^{16}	65536
2^{20}	1048576
2^{30}	1073741824
2^{40}	1099511627776
2^{50}	1125899906842624

Figure 1: Powers of two

Prefix	Name	Multiplier
d	deca	$10^1 = 10$
h	hecto	$10^2 = 100$
k	kilo	$10^3 = 1000$
M	mega	10^6
G	giga	10^9
T	tera	10^{12}
P	peta	10^{15}
E	exa	10^{18}
d	deci	10^{-1}
c	centi	10^{-2}
m	milli	10^{-3}
μ	micro	10^{-6}
n	nano	10^{-9}
p	pico	10^{-12}
f	femto	10^{-15}

Figure 2: SI system prefixes

Prefix	Name	Multiplier
Ki	kibi or kilobinary	2^{10}
Mi	mebi or megabinary	2^{20}
Gi	gibi or gigabinary	2^{30}
Ti	tebi or terabinary	2^{40}
Pi	pebi or petabinary	2^{50}

Figure 3: SI binary prefixes

Definition 1.2 (Properties of powers).

- *(Multiplication.)* $2^m \cdot 2^n = 2^m\, 2^n = 2^{m+n}$. *(Dot $\cdot$ optional.)*
- *(Division.)* $2^m/2^n = 2^{m-n}$. *(The symbol / is the slash symbol)*
- *(Exponentiation.)* $(2^m)^n = 2^{m \cdot n}$.

Example 1.1 (Approximations for 2^{10} and 2^{20} and 2^{30}). *Since* $2^{10} = 1024 \approx 1000 = 10^3$*, we have that* $2^{20} = \left(2^{10}\right)^2 \approx 1000^2 = 10^6$*, and likewise,* $2^{30} = \left(2^{10}\right)^3 \approx 1000^3 = 10^9$.

The last number, a one followed by nine zeroes, we call it a billion in American English; in (British) English a billion is a million millions (aka trillion). If one writes 10^9 or 10^{12} no confusion is possible; therefore avoid saying "billion" or might hear a joke about millions, billions and trillions.

Note 1.1. *A kilo uses a lower case* **k**. *A capital case* **K** *stands for Kelvin, as in degrees Kelvin.*

2 Logarithms base two (and e, and 10)

Definition 2.1 (Logarithm base two of n is $\lg(n)$). *The logarithm base two of n is formally denoted by $y = \lg(n)$ or if we drop the parentheses, $y = \lg n$, and is defined as the power y that we need to raise integer 2 to get n.*

$$\text{That is, } y = \lg(n) \iff 2^y = 2^{\lg(n)} = n.$$

From now on we will be using the informal form $y = \lg n$ without parentheses instead of $y = \lg(n)$. Another way to write both is $y = \log_2 n$ or $y = \log_2(n)$. The two writings: $\lg^k n = (\lg n)^k$ are equivalent. We sometimes write $\lg\lg n$ to denote $\lg(\lg(n))$ and the nesting can go on. Note that $\lg^{(k)} n$ with a parenthesized exponent means something else (it is the iterated logarithm function).

Definition 2.2 (Other Logarithms). *The other logarithms: $\log_{10}(x)$ or $\log_{10} x$ and $\ln(x)$ or $\ln x$ or $\log_e n$, are to the base 10 or to the base $e = 2.7172\ldots$ of the Neperian logarithms respectively. If one writes $\log n$, then the writing may be ambiguous. Note that if we tilt towards calculus we use x as in $\lg(x)$ but if we tilt towards computing or discrete mathematics we use n as in $\lg(n)$ for the indeterminate's i.e. variable's name.*

Expression $\lg(n)$	**Value** y	**Explanation** since $2^y = 2^{\lg n} = n$ (**by definition**)
$\lg(1)$	0	since $2^0 = 1$
$\lg(2)$	1	since $2^1 = 2$
$\lg(256)$	8	since $2^8 = 256$
$\lg(1024)$	10	since $2^{10} = 1024$
$\lg(1048576)$	20	since $2^{20} = 1048576$
$\lg(1073741824)$	30	and so on

Figure 4: Logarithms: Base two

Example 2.1. $\lg 2$ *is one since* $2^1 = 2$. $\lg(256)$ *is 8 since* $2^8 = 256$. $\lg(1)$ *is 0 since* $2^0 = 1$.

Theorem 2.1 (Properties of Logarithms.). *In general, $2^{\lg(n)} = n$ and thus,*

i. (Multiplication.) $\lg(n \cdot m) = \lg n + \lg m$.

ii. (Division.) $\lg(n/m) = \lg n - \lg m$.

iii. (Exponentiation.) $\lg(n^m) = m \cdot \lg n$.

iv. (Change of base.) $n^{\lg m} = m^{\lg n}$. *Moreover* $\lg a = \frac{\log a}{\log 2}$ *(whatever the base of the latter logs).*

Example 2.2. *Since* $2^{20} = 2^{10} \cdot 2^{10}$ *we have that* $\lg(2^{20}) = \lg(2^{10} \cdot 2^{10}) = \lg(2^{10}) + \lg(2^{10}) = 10 + 10 = 20$. *Likewise* $\lg(2^{30}) = 30$. *Drawing from the exercise of the previous page,* $\lg(1{,}000) \approx 10$, $\lg(1{,}000{,}000) \approx 20$ *and* $\lg(1{,}000{,}000{,}000) \approx 30$.

3 Sets and Sequences

Definition 3.1 (Set). *A set is a collection of elements in no particular order.*

Note 3.1 (Curly braces for a set)**.** *For a set we use curly braces* $\{$ *and* $\}$ *to denote it. In a* **set** *the order of its elements does not mater. Thus set* $\{10,30,20\}$ *is equal to* $\{10,20,30\}$*: both represent the same set containing elements 10,20, and 30 and thus* $\{10,30,20\} = \{10,20,30\}$.

Definition 3.2 (Sequence). *A sequence is a collection of elements ordered in a specific way.*

Note 3.2 (Angular brackes for a sequence)**.** *For a sequence we use angular brackets* $\langle$ *and* $\rangle$ *to denote it. In a sequence the order of its elements matters. Thus by using angular bracket notation sequence* $\langle 10,30,20\rangle$ *represents a sequence where the first element is a 10, the second a 30 and the third a 20. This sequence is different from sequence* $\langle 10,20,30\rangle$*. The two are different because for example the second element of the former is a 30, and the second element of the latter is a 20. Thus those two sequences differ in their second element position. (They also differ in their third element position anyway.) Thus* $\langle 10,30,20\rangle \neq \langle 10,20,30\rangle$.

Note 3.3 (Set vs Sequence)**.** *Sets include unique elements; sequences not necessarily. The* $\{10,10,20\}$ *is incorrect as in a set each element appears only once. The correct way to write this set is* $\{10,20\}$*. For a sequence repetition is allowed thus* $\langle 10,10,10\rangle$ *is OK.*

Note 3.4 (Set/Sequence with many elements)**.** *Sets or sequences with too many elements to write down: three periods (...). Thus* $\{1,2,\ldots,n\}$ *would be a way to write all positive integers from 1 to n inclusive. The three period symbol ... is also known as ellipsis (or in plural form, ellipses).*

4 Bits and bytes

Note 4.1 (Joke: 'Bits and bytes' capitalization)**.** *The capitalization in the section header is English grammar imposed and intended as an unintentional joke! Read through the end of this page in order to get it!*

Definition 4.1 (**Bit**)**.** *The word* **bit** *is an acronym derived from* **bi**`nary` `digi`**t** *and it is the minimal amount of digital information. The correct notation for a bit is a fully spelled lower-case* **bit***.*

A bit can exist in one of two states: `1` and `0`, or `High` and `Low`, or `Up` and `Down`, or `True` and `False`, or `T` and `F`, or `t` and `f`. **A lower-case** ***b*** **should never denote a bit!** Several publications mistakenly do so, however! If we want to write down in English 9 binary digits we write down `9bit`; a transfer rate can be 9.2bit/s or also 9.2bps. The notation `9b` should be considered **nonsense**.

Definition 4.2 (**Byte**)**.** *A* **byte** *is the minimal amount of binary information that can be stored into the memory of a computer and it is denoted by a capital case* **B***.*

Definition 4.3 (**Word**)**.** *Word is a fixed size piece of data handled by a microprocessor. The number of bit or sometimes equivalently the number of bytes in a word is an important characteristic of the microprocessor's architecture.*

Etymologically, a `byte` is the smallest amount of data a computer could `bite` out of its memory! We cannot store in memory a single bit; we must utilize a byte thus wasting 7 binary digits. Nowadays, **1B** is equivalent to 8bit. Sometimes a byte is also called an **octet**. A 32-bit architecture has word size 32 bit.

Definition 4.4 (**Memory size**)**.** **Memory size** *is usually expressed in bytes or its multiples.*

We never talk of 8,000bit memory, we prefer to write 1,000B rather than 1,000byte, or 1,000Byte.

Prefix	Name	Multiplier
1KiB	1kibibyte	$2^{10}B$
1MiB	1mebibyte	$2^{20}B$
1GiB	1gibibyte	$2^{30}B$
1TiB	1tebibyte	$2^{40}B$
1PiB	1pebibyte	$2^{50}B$

Figure 5: SI aggregates of a byte

Name	Multiplier
1 short	2B = 16bit
1 word	4B = 32bit
1 double word	8B = 64bit

Figure 6: Other aggregates of a byte

Definition 4.5 (**Confusing Notation: How many bytes in 1kB or 1MB or 1GB of RAM or Disk?**)**.** *In SI, 1kB implies 1,000B; likewise 1MB is 1,000,000B and 1GB is 1,000,000,000B. When we refer to memory (eg. RAM i.e. Random Access Memory or main memory), companies such as Microsoft or Intel mean that 1kB is 1,024B, that 1MB is 1,048,576B and 1GB is* $2^{30}B$*. To add to this confusion, hard disk drive manufacturers in warranties, define a 1kB, 1MB, and 1GB as in SI (1000B,* 10^6B *and* 10^9B *respectively).*

Example 4.1 (**When is 500GB equal to 453GB for the correct 453GiB?**)**.** *A hard-disk drive (say, Seagate) with 500GB on its packaging, will offer you a theoretical 500,000,000,000B. However this is unformatted capacity; the real capacity after formatting would be 2-3% less, say 487,460,958,208B. Yet an operating system such as Microsoft Windows 7 will report this latter number as 453GB. Microsoft would divide the 487,460,958,208 number with 1024*1024*1024 which is 453.93GiB i.e Microsoft's 453GB.*

Conclusion: Stick to KiB, MiB, GiB and avoid kB,MB,GB.

5 Notation

Some preliminaries.

Definition 5.1 (colon symbol : and pipe symbol $|$)**.** *The colon symbol : stands for* **such that***. The pipe symbol $|$ also stands for* **such that** *or alternatively for* **where***.*

Definition 5.2 (universal quantifier $\forall$)**.** *The $\forall$ symbol is also known as the* **universal quantifier***. It reads as* **for all***.*

Definition 5.3 (existential quantifier $\exists$)**.** *The $\exists$ symbol is also called the* **existential quantifier***. It reads as* **there exists** *in singular or in plural as* **there exist***.*

Definition 5.4 (set membership)**.** *Symbol $\in$ is the* **belongs to** *set membership symbol.*

Definition 5.5 (implication)**.** *$X \Rightarrow Y$ is also known as implication and can be stated otherwise as "X implies Y". Y is then necessary for X, and X is sufficient for Y.*

Definition 5.6 (Unknown, Variable, Indeterminate)**.** *For $f(x)$ or $\log(x)$ the x inside the parentheses is what is traditionally known as an unknown. Compu-speak we might call it a "variable" but we have not yet defined a variable formally. In Math we also call it an indeterminate. We called it before a value (or an operand).*

An **operator** in mathematics and also in computing is a symbol that indicates an **operation**. The object of the operator and its operation is known as the **operand**. An operation denoted by an operator (and thus a function) can have one or more operands and is then known as a unary operator/operation (one operand), binary (two operands), etc.

Definition 5.7 (Unary operators and operations)**.** *A unary operator is a symbol that indicates a unary operation, i.e. the application of a mathematical or computing function on one (single) value. The value is known as the operand of the operator (and the corresponding operation).*

When a unary operator is used, it precedes (or surrounds) its operand.

Any one of the trigonometric functions such as *sin* is a unary operator. In $sin(x)$, the *sin* is a unary operator, the x is the operand and $sin(x)$ is the unary opeation that involves the application of the sine trigonometric function on operand x.

Thus **operator is a symbol**, the **operation is the mathematical or comuting function** implied by the symbol (operator) and **the object of the operation or operator is the operand**.

Another unary operator is the absolute value function $||$. Thus $|5|$ is a 5. We also have unary operators $+$ and $-$ to assign a positive or negative sign to a number. Thus in $+5$ and -5 the operator also precedes its corresponding single operand.

Besides **unary operators** we also have **binary operators** that denote a **binary operation**.

Definition 5.8 (Binary operators, and operations)**.** *A binary operator is a symbol that indicates a binary operation i.e. the application of a mathematical or computing function on two values. The two values are known as the operands of the operator and its corresponding operation. The first one (from the left) is known as the left operand and the second one as the right operand.*

The $+$, the plus-symbol, is the (additive) binary operator that indicates the operation known as addition. In $5+3$, the operator is the plus ($+$), the operation is addition as implied by the presence of the additive operator plus. Operation addition is a binary operation and requires two operands present. Numbers 5 and 3 are the two operands that will participate in the operation: 5 is the left operand and 3 the right operand. In this context, operation addition is a binary operation and $+$ is a binary operator because they require two operands. Previously, the plus-symbol indicated a unary operator and operation. The presence of one or two operands resolves the type of the operator/operation (unary vs binary).

Definition 5.9 (Operator overload)**.** *The same operator can indicate one or more operations: one unary and one binary. The presence of one or two operands, that is the context, can be used to resolve the type of the operator and its coresponding operation.*

In expression $5-3$ the dash-symbol (also known as the minus-symbol) $-$ denotes operation subtraction and it is a binary operator. Operation subtraction is then a binary operation. The plus-symbol and similarly the dash-symbol are also unary operators and each one indicates the sign assigned to an integer.

Definition 5.10 (Prefix, postfix and infix notation)**.** *A unary operator requires one operand, a binary operator two operands. In the former case the operator precedes the operand. In the latter case the operator can precede, follow or be in-between the operands. Thus* $+5\ 3$ *or* $5\ 3+$ *or* $5+3$ *indicate the same addition operation in prefix, postfix and infix notation. In all cases 5 is the left operand and 3 is the right operand.*

We are used to using infix notation in describing operations.

Definition 5.11 (Integer vs Real ideterminate names)**.** *In functions defined hereafter we will shall more often use n instead of x. Indeterminate n implies a non-negative or positive integer. Indeterminate x implies a real number. We describe a discrete math universe of non-negative integers.*

Definition 5.12 (Algorithms: Problem size, Input size and names)**.** *Indeterminate n will usually denote problem size or input size. Thus for a sequence of n keys, n represents the number of elements or keys in the sequence, the problem size and also to some degree the input size. For a 2d-array (aka matrix) $n \times n$, n represents the problem size denoting the number of its rows or columns i.e. its geometry or shape of the matrix. In this latter example the input size is the size of the matrix i.e. its number of elements which is n^2.*

Definition 5.13 (Sums and Sigma notation)**.** *The sum $a_0 + a_1 + \ldots + a_n$ can be represented in compact form as*

$$\sum_{i=0}^{i=n} a_i = \sum_{i=0}^{n} a_i = a_0 + \ldots + a_n$$

Variable i has values that vary between a smallest value as indicated under the sum's Sigma symbol and it is $i=0$ in this example and its largest value as indicated over the sum's Sigma symbol and it is $i=n$). It also assumes all integer number values between 0 and n (inclusive of the end points). The variable's name is available beneath the Sigma and can be omitted over it as shown in the second formulation of the sum. The terms of the sum are usually members of a sequence and in this case $a_0, a_1, \ldots, a_n$. The general member of the sequence is a_i as described in the sigma / sum formulation. If the sequence is simple such as $a_i = i$ instead of a_i we use directly i; likewise for $a_i = i^2$, $a_i = i^3$.

Definition 5.14 (Integer numbers)**.** *An* **integer number** *is a number that takes integer values. It can be positive, negative or zero. For a positive integer number we might or might not place a plus sign + before its magnitude. For a negative integer number we always place a negative sign − before its magnitude.*

Definition 5.15 (Non-negative integer numbers)**.** *A* **non-negative integer number** *can be positive or zero.*

Definition 5.16 (**Natural (integer) numbers**: unsigned integers)**.** *A* **natural (integer) number** *is an integer number that is a positive integer number. However this definition varies and it might also mean a non-negative integer number. We also call it an* **unsigned integer**.

Most numbers listed below would be natural numbers (one way or the other). When we start talking about negative numbers this will be made very clear (and the discussion will be brief).

Example 5.1. *Integer 13 is a positive integer and so is* $+13$*. Integer* -13 *is a negative integer. Integer 0 is neither positive nor negative. Ordinarily, there should be no sign preceding a 0.*

Definition 5.17 (**Integer Numbers: Signed Integers**)**.** *In general, a* **(signed) integer** *number can be positive, negative or zero.*

A zero does not have a sign. A 5 or $+5$ mean the same thing: a positive sign for 5. Then a -5 means a negative sign for five i.e. minus five.

Definition 5.18 (**Real Numbers: Floating-point Numbers**)**.** *A real number that includes integer digits, possibly a decimal point, and decimal digits is called a* `floating-point` *number.*

Thus 12.1 or 12.10 or $1.21 \cdot 10^1$ all represent the same real number.

Definition 5.19 (Exponential notation)**.** *A real number can be expressed in exponential notation in the form* $a \times 10^b$ *or equivalently as* aEb *or* aeb*.*

Thus 5.1×10^3 is $5.1e3$ or $5.1E3$.

Definition 5.20 (Magnitude of a number real or integer)**.** *The magnitude of an integer or real number is its absolute value.*

Example 5.2 (Magnitude vs value)**.** *For a negative number such as* -5 *its magnitude is 5 and its value is* -5*. Thus the 'we always place a negative sign* $-$ *before its magnitude' above makes sense.*

6 Frequency and the Time domain

Definition 6.1 (Time)**.** *The unit of time is the second and it is denoted by* `1s` *or also* `1sec` *but the latter is not SI compliant.*

Submultiples are $\texttt{1ms}, 1\mu\texttt{s}, \texttt{1ns}, \texttt{1ps}$ which are $10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}$ respectively of a second, and are pronounced millisecond, microsecond, nanosecond, and picosecond respectively. **Note that a millisecond has two ells.**

Definition 6.2 (Frequency)**.** *Frequency is the number of times an event is repeated in the unit of time. The unit of frequency is* `cycles` *per* `second` *or* **cycles/s** *or just* **Hz**. *The symbol for the unit of frequency is the Hertz, i.e.* **1Hz = 1cycle/s.**

Then 1kHz, 1MHz, 1GHz, and 1THz are 1000, $10^6, 10^9, 10^{12}$ cycles/s or Hz. Note that in all cases the `H` of a `Hertz` is CAPITAL CASE, never lower-case. (The z is lower case everywhere.)

Definition 6.3 (Time vs Frequency)**.** *The relationship between time (t) and frequency (f) is inversely proportional. Thus* $f \cdot t = 1$

Thus 5Hz, means that there are 5 cycles in a second and thus the period of a cycle is one-fifth of a second. Thus `f=5Hz` implies `t=1/5s=0.2s.`

Computer or microprocessor speed used to be denoted in MHz and nowadays in GHz. Thus an `Intel 80486DX` microprocessor of the early 1990s rated at 25MHz, used to execute 25,000,000 instructions per second; one instruction had a period or execution time of roughly $1/25,000,000 = 40\mu s$.

A moderm CPU rated at 2GHz allows instructions to be completed in $1/2,000,000,000 = 0.5ns$.

And note that in the 1990s and also in the 2010s retrieving one byte of main memory still takes 60-80ns.

Definition 6.4 (A nanosecond is (roughly) one foot!)**.** *In one nanosecond, light (in vacuum) can travel a distance that is approximately 1 foot. Thus* `1 foot is approximately ''1nanosecond''.`

7 Number systems: Denary, Binary, Octal, and Hexadecimal

When one writes down number `13`, implicit in its writing is that the number is an integer number base-10. For integer numbers base-10 we utilize ten digits to describe them (i.e. write them down). These ten digits are `0-9`. The **base** is formally known as the **radix**. Integer numbers or numbers in general can be written down in a variety of radixes (the plural of radix). The most popular radix is radix-10 i.e. base-10.

Definition 7.1 (Radix-10 or Base-10: denary notation)**.** *A number written down in radix-10, also known colloquially as base-10, is expressed in* **denary** *notation by utilizing the ten digits 0 through 9 to write it. One can explicitly indicate the radix by writing the radix in the form of a subscript next to the number.*

The **den** of denary is a corruption of **ten** i.e. 10 and it means base-10 or radix-10!

Example 7.1 (A denary integer)**.** *Formally we should read* `13` *as "base-10 integer 13" or "radix-10 integer 13". To indicate the radix explicitly we may write* 13_{10}*; then we can skip the "base-10 integer" or "radix-10 integer" wording. In all three cases thirteen is expressed in* **denary** *notation. The left-most non-zero digit is the* **most-significant digit (msd)***, the right-most digit is the* **least-significant digit (lsd)***. Thus for integer 13 in radix-10, the 1 is the most-significant digit and the 3 is the least-significant digit.*

Note 7.1 (Caution!). *Avoid the use of the term* **decimal** *to refer to a radix-10 or base-10 integer expressed in denary notation. The term decimal implies a decimal point i.e. we imply a real number expressed in denary notation such as 13.0 or 13.31!*

Definition 7.2 (Denary natural numbers in fixed-width)**.** *An n-digit radix-10 natural integer number x is denoted as* $x = x_{n-1}x_{n-2}\ldots x_0$*, where* $x_i \in \{0,\ldots,9\}$ *for all* $0 \le i < n$*. The most-significant digit is* x_{n-1} *and the least-significant digit is* x_0*. The magnitude of the number is*

$$|x| = \sum_{i=0}^{i=n-1} x_i \cdot 10^i.$$

The value of x is its magnitude i.e. $a = |x|$*.*

The definition can easily extend to integer numbers in general.

Definition 7.3 (Denary integer numbers in fixed-width)**.** *An n-digit radix-10 natural integer number x is denoted as* $x = sx_{n-1}x_{n-2}\ldots x_0$*, where* $x_i \in \{0,\ldots,9\}$ *for all* $0 \le i < n$*, and s is* $+$ *or empty to indicate a positive integer, empty for zero, or* $-$ *to indicate a negative integer. The most-significant digit is* x_{n-1} *and the least-significant digit is* x_0*, and s is the sign. The magnitude of the number is*

$$|x| = \sum_{i=0}^{i=n-1} x_i \cdot 10^i.$$

The value of a is $x = (-1) \cdot |x|$ *if the sign of x is a negative one i.e.* -1*, or its magnitude* $x = |x|$ *otherwise.*

Example 7.2 (Units of radix-10 integer)**.** *For* $x = x_{n-1}x_{n-2}\ldots x_0$*,* $x_i \in \{0,\ldots,9\}$ *for all* $0 \le i < n$*, the digit* x_i *indicates the number of times the corresponding multiplier* 10^i *is going to be used to derive the magnitude of the radix-10 integer.*

For the example above x_0 is the number of units, x_1 is the number of tens, x_2 is the number of hundreds, x_3 is the number of thousands contributing to the magnitude of x.

Method 7.1 (Finding the magnitude of a radix-10 integer)**.** *For* $a = 123456_{10}$ *we have, 6 units, 5 tens, 4 hundreds, 3 thousands, and so on. To derive its magnitude, we write all powers of* 10 *right to left from most to least significant digit over the number, multiply the corresponding digit and power and add up the results.*

$$
\begin{array}{lclclclclclcll}
10^5 & & 10^4 & & 10^3 & & 10^2 & & 10^1 & & 10^0 & & & \text{generate powers} \\
\cdot & + & \cdot & + & \cdot & + & \cdot & + & \cdot & + & \cdot & & & \\
1 & & 2 & & 3 & & 4 & & 5 & & 6 & = & & \text{digits} \\
1\cdot 10^5 & + & 2\cdot 10^4 & + & 3\cdot 10^3 & + & 4\cdot 10^2 & + & 5\cdot 10^1 & + & 6\cdot 10^0 & = & & \text{pairwise product} \\
100,000 & + & 20,000 & + & 3,000 & + & 400 & + & 50 & + & 6 & = & 123,456 & \text{add up results}
\end{array}
$$

Example 7.3 (Leading zeroes vs Trailing zeroes)**.** *Leading zeroes do not change the outcome. So 123456 and 00123456 are the same number; the two leading zeroes have no effect. Trailing zeroes are important, 123456 and 12345600 are two different numbers. The latter can be derived from the former by multiplying with* 10^2 *i.e. 10 raised to the number of trailing zeroes to derive the latter number from the former. This is also the case for 12300 and 1230000.*

	Base or Radix	# digits	digits
Binary	2	2	0 , 1
Octal	8	8	0 .. 7
Denary	10	10	0 .. 9
Hexadecimal	16	16	0 .. 9 , a .. f ; alternative: 0 .. 9 , A .. F

Fact 7.1 (Table of integers)**.** *A table of some integers in binary, octal, hexadecimal and denary is shown below.*

Binary	*Denary*	*Hexadecimal*	*Octal*	*Binary(4-bit)*
0	0	0	0	0000
1	1	1	1	0001
10	2	2	2	0010
11	3	3	3	0011
100	4	4	4	0100
101	5	5	5	0101
110	6	6	6	0110
111	7	7	7	0111
1000	8	8	10	1000
1001	9	9	11	1001
1010	10	A	12	1010
1011	11	B	13	1011
1100	12	C	14	1100
1101	13	D	15	1101
1110	14	E	16	1110
1111	15	F	17	1111

Shorthand
0o17 for octal 17
0xff for hexadecimal FF lower-case
0xFF for hexadecimal FF upper-ace

Sometimes
017 indicates 0o17

Definition 7.4 (Binary notation of a natural number)**.** *A natural number x is denoted in radix-2 as the n-digit (or n-bit) sequence $bin(x) = x_{n-1}x_{n-2}\ldots x_0$, where $x_i \in \{0,1\}$ for all $0 \le i < n$. For $bin(x)$ in binary notation its magnitude and value x is*

$$x = |x| = \sum_{i=0}^{i=n-1} x_i \cdot 2^i.$$

Definition 7.5 (Octal notation of a natural number)**.** *A natural number x is denoted in radix-8 as the n-digit sequence $oct(x) = x_{n-1}x_{n-2}\ldots x_0$, where $x_i \in \{0,1,2,3,4,5,6,7\}$ for all $0 \le i < n$. For $oct(x)$ in octal notation its magnitude x is*

$$x = |x| = \sum_{i=0}^{i=n-1} x_i \cdot 8^i.$$

Definition 7.6 (Hexadecimal notation of a natural number)**.** *A natural number x is denoted in radix-16 as the n-character sequence $hex(x) = x_{n-1}x_{n-2}\ldots x_0$, where $x_i \in \{0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f\}$ for all $0 \le i < n$, or equivalently $x_i \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$ for all $0 \le i < n$. For $hex(x)$ in hexadecimal notation its magnitude x is*

$$x = |x| = \sum_{i=0}^{i=n-1} x_i \cdot 16^i,$$

with an a or A being interpreted as ordinal 10, b or B as an 11, c or C as a 12, d or D as a 13, e or E as a 14, and f or F as a 15.

Example 7.4. *If we write 101 we might indicate 101_{10} or 101_2 or 101_8 or 101_{16}.*
101_{10} is 101 in denary which is one hunder and one.
101_2 is $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$ i.e. five in denary.
101_8 is $1 \cdot 8^2 + 0 \cdot 8^1 + 1 \cdot 8^0 = 65$ i.e. sixty-five in denary.
101_{16} is $1 \cdot 16^2 + 0 \cdot 16^1 + 1 \cdot 16^0 = 257$ i.e. two-hundred fifty seven in denary.

Binary into Denary

Example 7.5 (Convert binary into denary). *Find the magnitude or value x of the 5-bit binary number ($n = 5$) with $bin(x) = 11001$.*

$$\begin{array}{ccccccccccccc}
2^4 & & 2^3 & & 2^2 & & 2^1 & & 2^0 & & \\
\cdot & & \cdot & & \cdot & & \cdot & & \cdot & & \\
1 & & 1 & & 0 & & 0 & & 1 & = & \\
 & & & & & & & & & & \\
16 & + & 8 & + & 0 & + & 0 & + & 1 & = & 25
\end{array}$$

Thus $x = |x| = 25$.

All previous definition are for a fixed-width notation, where the number of digits used is fixed to n. It is possible then that leading zeroes would appear in the representation. Thus leading zeroes can be suppressed and all natural numbers in a given range say from 0 to $2^m - 1$ can be represented with a minimal number of m binary digits (aka bits).

Example 7.6 (Fixed-width vs Minimal-width).

$\mathbf{0}, \mathbf{1}$	represented in binary with	1 bit *as*	$0, 1$
$\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}$	represented in binary with	2 bits *as*	$00, 01, 10, 11$
$\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}$	represented in binary with	3 bits *as*	$000, 001, 010, 011, 100, 101, 110, 111$
$\mathbf{0} - \mathbf{15}$	represented in binary with	4 bits *as*	$0000 - 1111$
$\ldots$			
$0 \ldots 2^m - 1$	represented in binary with	m bits *as*	$\underbrace{00\ldots0}_{m\ bits} - \underbrace{1\ldots1}_{m\ bits}$

Example 7.7 (m bits for a natural number). *What is the range of natural numbers that can be represented with m bits? What is the smallest and largest natural number? How many natural numbers in total. The answer is* 2^m *as shown above with the smallest being* 0 *i.e. m zeroes and the largest* $2^m - 1$ *i.e. m ones.*

Example 7.8 (m ones). *The value a of m-bit* $bin(a) = \underbrace{1\ldots1}_{m\ ones}$ *is* $a = 2^m - 1$.

Example 7.9 (One followed by $m-1$ zeroes). *The value a of m-bit* $bin(a) = 1 \underbrace{0\ldots0}_{m-1\ zeroes}$ *is* $a = 2^{m-1}$.

Definition 7.7 (**Octal Notation: Addendum**). *Instead of writing* 101_8 *we can also write* $0o101$ *to indicate a number in octal notation. (In the past a leading zero indicated an octal number but this is confusing: 0o101 is more obvious than an 0101.)*

Definition 7.8 (**Hexadecimal Notation: Addendum**). *Instead of writing* 101_{16} *we can also write* $0x101$ *to indicate a number in hexadecimal notation. Moreover one can use* $0x$ *or* $0X$ *depending on whether* $a - f$ *or* $A - F$ *are to be used in the representation: thus we write* `0x1f1` *for the lower-case f's appearance and* `0X1F1` *for the upper-case F's appearance.*

Example 7.10 (**n-bit or n-digit natural numbers**). *An n-bit notation indicates a binary number representation as a* **bit** *is an indicator of a* binary digit*. Otherwise we use the term n-digit and the word digit can also refer to the a-f or A-F of a number in hexadecimal notation.*

Example 7.11.

```
101        is    radix-10, radix-2, radix-8, radix-16 representation of 101,  5, 65, 257 resp.
81  cannot be in           radix-2, radix-8           because it uses an 8 (radix-10, radix-16 only)
AB  cannot be in radix-10, radix-2, radix-8,          becaused digits A,B are radix-16 digits only
```

Fact 7.2 (Number of bits for unsigned integer $a > 0$). *The number of bits m without leading zeroes in bin(a) of a natural number a is given by* $m = \lfloor \lg a \rfloor + 1 = \lceil \lg (a+1) \rceil$.

Thus for 1 we need 1 bit, for 2 i.e. 10_2 we need two, for 4 ie. 100_2 we need three and for 7 i.e 111_2 we also need three.

Proof. From Example 7.8 and Example 7.9 we have that any natural number a such that $2^{m-1} \leq a \leq 2^m - 1$ needs m bits for its representation $bin(a)$. Natural numbers $a < 2^{m-1}$ need $m-1$ or fewer bits; they can become m-bit by using leading zeroes as shown in Example 7.6.

The range of a with leading bit one $2^{m-1} \leq a \leq 2^m - 1$ can be rewritten as $2^{m-1} \leq a < 2^m$. Taking logarithms base two we have $m-1 \leq \lg a < m$.

$$
\begin{aligned}
2^{m-1} \leq a & \leq 2^m - 1 \\
2^{m-1} \leq a & < 2^m \\
m-1 \leq \lg(a) & < m. \\
m-1 \leq \lfloor \lg(a) \rfloor \leq \lg(a) & < m.
\end{aligned}
$$

Since $\lfloor \lg(a) \rfloor \leq \lg(a)$ and by the last inequality above less than m, we have that consecutive integers $m-1$ and m are the only two integers surrounding $\lg(a)$. If $\lfloor \lg(a) \rfloor \leq \lg(a)$ cannot be m it should be $m-1$ i.e. $m-1 = \lfloor \lg a \rfloor$ implying $m = \lfloor \lg a \rfloor + 1$.

Similarly,

$$
\begin{aligned}
2^{m-1} \leq a & \leq 2^m - 1 \\
2^{m-1} + 1 \leq (a+1) & \leq 2^m \\
2^{m-1} < (a+1) & \leq 2^m \\
m-1 < \lg(a+1) & \leq m. \\
m-1 < \lg(a+1) \leq \lceil \lg(a+1) \rceil & \leq m.
\end{aligned}
$$

Since $\lceil \lg(a+1) \rceil > m-1$ and $\lceil \lg(a+1) \rceil \leq m$, there can be only one possibility that $\lceil \lg(a+1) \rceil = m$. □

Extension of Algorithm in Example 7.5 : Radix-b to Radix-10

Fact 7.3 (Radix-b to Radix-10t). *We can convert a radix-b into radix-10 either left-to-right or right-to-left. The example below is left-to-right for $b = 2$.*

```
Algorithm Base-b-to-Base-10          The example shown is for binary to decimal conversion
Algorithm Base-2-to-Base-10          RES*b +  t  -> RES    > shows bit that is being read
                                                             that is the t value
RES=0; b=2;                              0    -      0         1  0  1  0  1  1

repeat until all bits are read        0*b +  1  -> 1         >1  0  1  0  1  1
  read_next_bit t;  %shown next to .  1*b +  0  -> 2          1 >0  1  0  1  1
  RES = RES * b +t; % RES next to  =  2*b +  1  -> 5          1  0 >1  0  1  1
                                      5*b +  0  -> 10         1  0  1 >0  1  1
                                     10*b +  1  -> 21         1  0  1  0 >1  1
                                     21*2 +  1  -> 43         1  0  1  0  1 >1
```

Binary into Octal, Binary into Hexadecimal

Fact 7.4 (**Radix-2 to Radix-8: Groups of 3 bit).** *For natural number a for which bin(a) is available, it octal notation can be derived easily by grouping bits into groups of three right to left and converting the three-bit binary into the corresponding octal digit using the Table of Fact 7.1. (The leftmost group might have its binary digits padded with leading zeroes to have four bits.)*

```
' 11'111'111    : Group into groups of 3 bits   : Step 1
'011'111'111    : Add leading zeroes left group : Step 2
  3   7   7     : Convert triplets into octal   : Step 3  [Use also Table of Fact 6.1]
   0o377        : Output                        : Step 4
```

Fact 7.5 (**Radix-2 to Radix-16: Groups of 4 bit).** *For natural number a for which bin(a) is available, it hexadecimal notation can be derived easily by grouping bits into groups of four right to left and converting the four-bit binary into the corresponding hexadecimal digit using the Table of Fact 7.1. (The leftmost group might have its binary digits padded with leading zeroes to have three bits.)*

```
'1111'1111      : Group into groups of 4 bits   : Step 1.
'1111'1111      : Add leading zeroes left group : Step 2
   F    F       : Convert quadruplets into hex  : Step 3  [Use also Table of Fact 6.1]
   0XFF         : Output using A-F              : Step 4
   or
   0xff         : Output using a-f              : Step 4
```

Binary into Octal, Binary into Hexadecimal

Fact 7.6 (Radix-10 to Radix-2: Right to Left).

Input : *Decimal integer a.*
Output: *Binary representation of bin(a) of a. (Right to left.)*

Step 1. *Set $X = a$. Bit sequence will be generated right-to-left, least-to-most significant.*
Step 2. *If X is even, generate a 0, set $X = X/2$, and Go to Step 4; otherwise (X odd) go to Step 3.*
Step 3. *If X is odd, generate a 1 and set $X = (X-1)/2$. Go to Step 4.*
Step 4. *If X is 0 go to Step 5, else go to Step 2 and repeat.*
Step 5. *Output the result (write it down properly).*

Fact 7.7 (Radix-10 to Radix-2: Left to Right).

Input : *Decimal integer a.*
Output: *Binary representation of bin(a) of a. (Left to right.)*

Step 1. *Starting with 1, compute by doubling $2^0, 2^1, \ldots, 2^m$ the largest $2^m \leq a$. Set $X = a$. $P = 2^m$.*
Step 2. *If X is equal to 0 Go to Step 5 else continue to Step 3.*
Step 3. *If $X \leq P$ output '1' , set $X = X - P$, $P = P/2$. Go to Step 2.*
Step 4. *If $X > P$ output '0' , set $P = P/2$. Go to Step 2.*
Step 5. *Done.*

Definition 7.9 (bin(x,n)). *We shall use the notation* $\mathbf{bin}(\mathbf{x},\mathbf{n})$ *to denote the n-bit notation of denary x in binary. Leading zeroes are used to pad the result to n bit or n ignored if the result has more than n bit. An alternative formulation* $\mathbf{bin}(\mathbf{x},\mathbf{n},\mathbf{m})$ *allows x to be in arbitrary radix m. (See den(x,n,m) for more explanation.)*

Definition 7.10 (oct(x,n)). *We shall use the notation* $\mathbf{oct}(\mathbf{x},\mathbf{n})$ *to denote the n octal-digit notation of denary x in octal representation. Leading zeroes are used to pad the result to n digit octal or n ignored if the result has more than n digit octal. An alternative formulation is* $\mathbf{oct}(\mathbf{x},\mathbf{n},\mathbf{m})$.

Definition 7.11 (hex(x,n) or HEX(x,n)). *We shall use the notation* $\mathbf{hex}(\mathbf{x},\mathbf{n})$ *or* $\mathbf{HEX}(\mathbf{x},\mathbf{n})$ *to denote the n hexadecimal-digit notation of denary x in hexadecimal representation. Leading zeroes are used to pad the result to n digit hexadecimal or n ignored if the result has more than n hex digits. An alternative formulation is* $\mathbf{hex}(\mathbf{x},\mathbf{n},\mathbf{m})$. *If* $\mathbf{HEX}(\mathbf{x},\mathbf{n})$ *or* $\mathbf{HEX}(\mathbf{x},\mathbf{n},\mathbf{m})$*, they both use capital case characters in the output.*

Thus bin(2,8)= 00000010 and bin(0,4)=0000. But note that bin(2,1)= 10. Moreover oct(2,5)= 00002, oct(8,2)= 10 and hex(10,2)= 0a whereas HEX(10,2)=0A.

Note that when we use the notation oct(8,2) we write oct(8,2)=10 rather than oct(8,2)=0o10. The octal number is not standalone but the result (value) of a function's application!

Definition 7.12 (den(x,n) or den(x,n,m)). *Function* $\mathbf{den}(\mathbf{x},\mathbf{n},\mathbf{m})$ *converts x into n-digits denary. If n is insufficient (too small) it generates as many digits as needed with no leading zeroes. By default, for* $\mathbf{den}(\mathbf{x},\mathbf{n})$*, x is in binary unless it is preceded by* 0*x or* 0*X or* 0*o, and then it is in hexadecimal or octal. Function* $\mathbf{den}(\mathbf{x},\mathbf{n},\mathbf{m})$ *has x in binary if* $m=2$*, in octal if* $m=8$*, and hexadecimal if* $m=16$*, and in that case x does not require a prefix.*

Thus den(10,0)=2, den(1000,2)=08, den(10,0,2)=2, den(10,0,8)=8, den(10,0,16)=16. Likewise, den(0o10,0)=8, den(0x10,0)=16. bin(x,n) is bin(x,n,10) and similarly for the other ones.

Operating Systems : Page numbers and offsets.

In operating systems, sometimes we need to do bit manipulations or extraction of information. A logical address referring to memory in general is an integer between 0 and $n-1$. The size of memory is n. In practice n is a power of two. This means that all memory addresses from 0 through $n-1$ can be represented with the same fixed number of binary digits needed for the representation of the largest integer in the range, $n-1$. By way of Fact 7.2 this is $\lg n$ if we substitute $a=n-1$ in Fact 7.2, and given that n is (assumed to be) a power of two no ceilings or floors are needed.

In operating systems a flat logical memory space of n bytes is split into pages of equal size. The size of a page is s and s is also a power of two. Thus an n address space can be split into $G=n/s$ pages, each of size s bytes.

Divisions involving powers of two become subtractions i.e. shift-right operations. The fact that both n and s are powers of two helps a lot. Division (as in n/s) is exact with no decimal (i.e. quotient is integer and remainder is zero). Moreover we can avoid division by subtracting the exponents. Thus dividing 256 by 8 the regular way requires a division but dividing 2^8 with 2^3 requires a subtraction between the exponents 8 and 3 i.e. $8-3=5$. The result is $2^{8-3}=2^5$ i.e. 32. In fact we can avoid even that subtraction if we maintain the original numbers and the results in binary : 256 in minimal binary representation (no leading zeroes) is 100000000 and 8 is 1000. Division of $256=2^8$ by $8=2^3$ is equivalent to shifting the binary representation of 256 three positions to the right (i.e. we shift the Dividend 256 three positions to the right, with three being the exponent of the divisor with the result being the quotient i.e. a 100000 which is binary for 32.)

```
    Divisor                  Dividend    ;   Q= Divisor / Dividend
          (Both are power of two)
    256   integer-divison-by     8                        in-denary
    2**8                     2**3                         in-denary but base 2 expo notation
Q=  2**8          /          2**3 = 2**(8-3)= 2**5  = 32 in denary

    bin(256)=100000000       bin(8)= 1000                 in-binary
Q=  SHIFTRIGHT(100000000,3)=100000            Q is 100000 in binary (i.e. 2**5)
```

Convert a logical address to a page number and an offset: $L = (P,T)$**.** A logical memory address L in the range 0 to $n-1$ can be expressed then as a page number P and offset T within a page: $L = (P,T)$. If the number of pages is G then P varies from 0 to $G-1$. If page size is s bytes, T varies from 0 to $s-1$.

Logical address L gets mapped to pair (P,T) of a page number P, and an offset T, where $0 \le L < n$, $0 \le P < n/s$, and $0 \le T < s$.

There is an easy way to obtain P and T from L: $P = \textbf{floor}(L/s)$, $T = L \bmod s$. Function mod is denoted in C/C++ by the % sign to denote the integer remainder when dividing the left hand side with the right hand side. The left-hand size (L) is the dividend, and the right-hand side (s) is the divisor of the division. The quotient is P and the remainder of the division is T (the offset).

Definition 7.13 (**Convert a logical address to a page number with offset: L=(P,T)).** *A memory space of n bytes, supports a paging system of page size s bytes. A logical (absolute) address L in that memory space can be mapped into a page P and offset T within that page:* $L = (P,T)$*. The mapping is as follows:*

$$(n,s): \quad L = (P,T) \quad \Rightarrow \quad P = \textbf{floor}(L/s), \quad T = L \bmod s, \qquad 0 \le L < n,\ 0 \le P < n/s,\ 0 \le T < s.$$

In C/C++ **floor** *is integer division and* mod *is denoted %. Thus another way to write it is to say*

$$(n,s): \quad L = (P,T) \quad \Rightarrow \quad P = (L/s), \quad T = L\%s, \qquad 0 \le L < n,\ 0 \le P < n/s,\ 0 \le T < s.$$

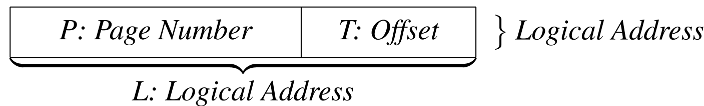
Definition 7.14 (**Convert a page number with offset (P,T) into a logical address** L**.).** *Moreover, given* (P,T) *we can recover L if we know the page size s.* **From** (P,T) **to L.**

$$(n,s): \quad (P,T) = L \quad \Rightarrow \quad L = P \times s + T, \qquad 0 \le L < n,\ 0 \le P < n/s,\ 0 \le T < s.$$

Example 7.12. *(To make initial calculations easy, we drop the power of two requirement.) If we have a memory of size* $n = 100,000B$ *and a paged organization with page size* $s = 5,000B$*, then we can view memory as a collection of 20 pages (*$n/s = 100000/5000 = 20$*) each of size* $s = 5000B$*. Thus an* $L = 23456$ *gets mapped to* $P = 23456/50000 = 4$*, and* $T = 23456\%5000 = 3456$*. Therefore* $L = (P,T)$ *is* $23456 = (4,3456)$*. Moreover we can retrieve L from* (P,T)*:* $L = P \times s + T$*. Therefore* $23456 = 4 \times 5000 + 3456$*.*

In binary, most information about s and $G = n/s$ can be retrieved from the bit sequence representing L.

Definition 7.15. *We view a logical address L as the concatenation of a page number P and and an offset T. Thus $L = (P,T)$ becomes $L = \langle P,T \rangle$, where $\langle\rangle$ is the concatenation operator.*



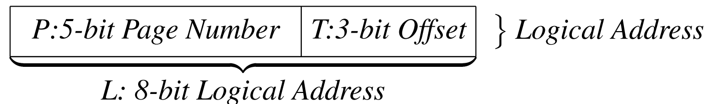
Suppose that $n = 256$. Then $\lg n = 8$ and we use 8-bit addresses.

Suppose that $s = 8$. Then $\lg s = 3$.

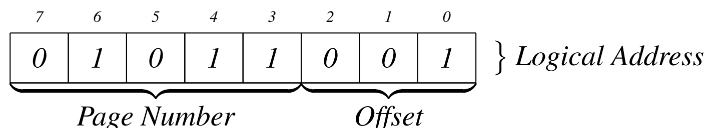
In this case $G = n/s = 32 = 2^5$ i.e. $0 \leq P < 2^5 = G$. Thus we need $\lg G = \lg 32 = 5$ bit for the page number P.

Moreover since $s = 8$, we have that $0 \leq T < 2^3 = s$. Thus we need $\lg s = \lg 8 = 3$ bit for the offset T.

Definition 7.16. *An 8-bit logical address L is thus the concatenation of the 5-bit page number P and a 3-bit offset T. Thus $L = (P,T)$ becomes $L = \langle P,T \rangle$, where $\langle\rangle$ is the concatenation operator.*



Example 7.13. *We can easily extract all relevant information from the picture below. Memory space has $n = 256$ bytes. Number of bits is 8 since $n = 256$ and thus $\lg n = 8$. A logical address L is in the range $0 \leq L < n = 256$ and thus needs 8 bit for its representation. Given that the page size s is 8B an offset T needs $\lg s = \lg 8 = 3$ bit and thus $0 \leq T < s = 2^3 = 8$. Moreover $G = n/s = 256/8 = 32$ and $\lg G = 5$ and thus a page number P is 5-bit since $G = 2^5$ and therefore $0 \leq P < G = 2^5 = 32$.*



Let $L = 01011001$ be in binary. The logical address is the binary 01011001 which is 89 in denary. The page number P is the binary 01011, the left-most five bit of L. In denary, this is 11. Thus $P = 11$. The offset T is the binary 001, the right-mist three bit of L. In denary, this is 1. Thus offset $T = 1$. Because n, s are powers of two an arbitrary L in the range $0 \ldots n-1$ can be mapped into (P,T) without a division but with just bit manipulation. Of course we could have extracted (P,T) from L using integer division by establishing a quotient (which is P) and a remainder (which is T) from the dividend L and the divisor s: $(L/s, L\%s) = (89/8, 89\%8) = (11,1)$.
Moreover $L = P \times s + T = 11 \times 8 + 1 = 89$.

8 ASCII, Unicode, UTF-8, UTF-16

Sequences of bits (or bytes) can be viewed as an unsigned integer (positive or non-negative integer), or signed integer (positive or negative or zero), or a real number (fixed-point or floating-point). They can also be viewed as the representation of a symbol (also known as 'character') in a string. A symbol (character) can be a letter in a language (eg. English, Greek, Central European, Chinese, etc), a digit, a punctuation mark or any other special (auxiliary) symbol. For example, the byte in Example 8.1 and also in Example 8.2 could represent natural number 65 in 8-bit and 16-bit binary notation. It is also the ASCII (American Standard Code for Information Interchange) representation of the letter `A` in English in Example 8.1 and the Unicode representation of the same letter `A`.

Example 8.1.

7	6	5	4	3	2	1	0	
0	1	0	0	0	0	0	1	} *ASCII for A*

Example 8.2.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	} *Unicode for A*

Example 8.3.

7	6	5	4	3	2	1	0	
0	0	1	1	0	0	0	1	} *ASCII for 1*

Example 8.4.

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	1	} *8-bit representation of natural number 1*

Fact 8.1 (ASCII)**.** *An english letter or a digit or a punctuation mark, or any other auxiliary symbol is represented in ASCII as a 7-bit bit-sequence and stored in a single byte. The corresponding numeric value is known as the ordinal (value) of the character. ASCII is limited to representing 128 symbols (with 'extensions' to represent up to 256 symbols.)*

Example 8.5 (ASCII and the first character of the alphabet)**.** *The ASCII representation of the upper-case english letter A is* `1000001`*. The byte view containing it is shown in Example 8.1. The ordinal value of that byte, viewed as an unsigned integer, is 65.*

Example 8.6 (ASCII and the digit one)**.** *The ASCII representation of the symbol that is numeric digit one (*1*) is* `0110001`*. The byte view containing it is shown in Example 8.3. The ordinal value of that byte, viewed as an unsigned integer, is 49. Natural number one (*1*) represented as a numerical value has the 8-bit representations shown in Example 8.4. Thus symbol 1 has a different ordinal value than the magnitude of the binary representation of natural number one.*

Fact 8.2 (Table of ASCII characters)**.** *The table below contains the ASCII representation of all 128 ASCII symbols arranged in 8 rows (0-7 in octal or hexadecimal) of 16 columns (0-F in hexadecimal). The ASCII code (ordinal value) for a character in hexadecimal notation can be retrieved by concatenating the row index (code) with the column index code.*

For example `A` is in row `4` and column `1` i.e. its hexadecimal code is `0x41`. Converting radix-16 into radix-10 we get 65 the ordinal value for *A*. Its row index `4` in 4-bit binary is `0100` and `1` in 4-bit binary is `0001`. Thus the code for `A` is `01000001` which is `65` in decimal or `0x41` in hexadecimal. Rows 0 and 1 contain Control Characters represented by the corresponding mnemonic code/symbol. Code `32` or `0x20` is the space symbol (empty field).

Fact 8.3 (Unicode Standard)**.** *The Unicode Standard uses two or more bytes to represent one symbol (character). Ordinal values in Unicode are known as code-points. The characters from* `U+0000` *to* `U+FFFF` *form the Unicode Standard basic multilingual plane (BMP). Characters with code-points higher than* `U+FFFF` *are called supplementary characters. The Unicode character for an ASCII character remains the same if one adds extra zeroes (padding). Thus the Unicode representation for an ASCII character is a zero-bit byte followed by a byte of the ASCII representation.*

Example 8.2 shows the Unicode representation of letter `A`. The first byte is a zero-bit byte followed essentially by the ASCII byte for `A`. Likewise, symbol `DEL` which is `0x7F` in ASCII has Unicode representation (code) `0x007F`. We also write this as `U+007F`.

Fact 8.4 (Java char)**.** *In Java the* **char** *data type has size 2B; java uses UTF-16 representation. It can only reprent and represents the Unicode Standard basic multilingual plane (BMP) that is the characters from* `U+0000` *to* `U+FFFF`*. Its minimum code-point is '\u0000' (or* `U+0000`*) and its maximum code-point is '\uFFFF' (or* `U+FFFF`*).*

There are several encoding to represent Unicode symbols. One of them is UTF-8 where symbols are encoded using 1 to 6 bytes. The UTF-8 representation of an ASCII symbol is the ASCII representation of that symbol for compatibility reasons and also for space efficiency. Another one is UTF-16 employed by Java.

Fact 8.5 (UTF-8)**.** *UTF-8 encodes characters in 1 to 6 bytes.*

- *ASCII symbols with ordinal values 0-127 are also Unicode symbols* `U+0000` *to* `U+007F` *and are represented in UTF-8 encoded as byte 0x00 to 0x7F; the seven least-significant bits of a byte is the ASCII code for the symbol with the most-siginificant bit being a zero.*
- *Unicode symbols with ordinal values larger than* `U+007F` *use two or more bytes each of which has the most significant bit set to 1.*
- *The first byte of a non-ASCII character is one of* `110xxxxx, 1110xxxx, 11110xxx, 111110xx, 1111110x` *and it indicates how many bytes there are altogether or the number of 1s following the first 1 and before the first 0 indicates the number of bytes in the rest of the sequence. All remaining bytes other than the first start with* `10yyyyyy`.

Example 8.7 (UTF-8, ASCII, Unicode)**.**

- *ASCII and UTF-8 encoding look the same.*
- *No ASCII code can appear as part of any other UTF-8 encoded Unicode symbol since only ASCII characters have a 0 in the most-significant bit position of a byte.*

Fact 8.6 (UTF-16). *UTF-16 is a character encoding that use one or two 16-bit binary sequences to encode all 1,112,604 code points of Unicode. The characters from BMP are presented with 2B (i.e. one 16-bit binary sequence), the surrogates with 4B.*

ASCII CHARACTER SET

\	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

NUL = null	BS = Backspace	DLE = Datalink escape	CAN = cancel
SOH = start of heading	TAB = horizontal tab	DC1 = Device control 1	EM = end of medium
STX = start of text	LF = linefeed/newline	DC2	SUB = substitute
ETX = end of text	VT = vertical TAB	DC3	ESC = escape
EOT = end of transmission	FF = form feed/newpage	DC4	FS = file separator
ENQ = enquiry	CR = carriage return	NAK = negative ACK	GS = group separator
ACK = acknowledge	SO = shift out	SYN = synchronous idle	RS = record separator
BEL = bell	SI = shift in	ETB = end of trans. block	US = unit separator

UTF-8 ENCODING

UTF-8	Number of bits in code point	Range
0xxxxxxx	7	00000000-0000007F
110xxxxx 10xxxxxx	11	00000080-000007FF
1110xxxx 10xxxxxx 10xxxxxx	16	00000800-0000FFFF
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	21	00010000-001FFFFF
111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	26	00200000-03FFFFFF
1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	31	04000000-FFFFFFFF

9 Signed Integers and Floating-point numbers: binary notation

As we mentioned earlier in the previous section, a byte or a collection of bytes (e.g. a word) can be viewed as the binary representation of a natural number or an ASCII symbol, or a Unicode Standard symbol, or UTF-8 or UTF-16 that encodes Unicode symbols. (And ASCII symbols are part of Unicode as well.)

Of interest in this section is the representation of not just natural numbers (positive or non-negative integer numbers) but of integer numbers in general: positive, negative, or zero. We call the latter signed integers to stress that they include all three groups.

Fixed-width. We describe some fixed width methods that represent signed integers with one, two, or four bytes: they can be extended to any fixed number of bytes, e.g. eight.

Fact 9.1 (N byte signed integers)**.** *If we were given N bytes i.e. $8N$ binary digits the number of positive, negative and zero values that can be represented is an even number and equal to 2^{8N}. If the number of positive integer values that can be represented is p, the number of negative values is n and there is a single zero, then $n+p+1 = 2^{8N}$ implies that $n+p$ must be an odd number: we cannot represent the same number of positive and negative values, unless we have more than one representation of zero.*

Example 9.1 ($N = 1$: 8-bit representation)**.** *If we use 1B, which is 8 bit, to represent a natural number (i.e. unsigned integer), we can represent with that byte $2^8 = 256$ consecutive numbers from 0 to 255.*
If we try to represent an integer (i.e. signed integer) we need to think about the representation of the sign (positive or negative in one bit) and the representation itself. If we attempt to represent in binary $2^7 = 2^8/2 = 128$ negative values, the remaining values must represent the zero and no more that 127 positive values.

```
8-bit unsigned                    All integers from 0 to 255
8-bit signed (two's complement)   All integers from -128 to  -1 , 0 , 1 to 127
```

We present three representations of signed integers: signed mantissa, one's complement, and two's complement. All three of them use the leftmost bit as a sign bit indicator: one indicates a negative number and a zero a positive number.

Caution: We shall use the term leftmost bit and most-significant bit very carefully. In signed integer representation, the leftmost bit is a sign bit. The most significant bit of the number is the one to the right of the sign bit i.e. the second from left bit.

9.1 Unsigned Integers

Fact 9.2 (**n-bit unsigned integer**). *An n-bit unsigned integer N has*

- *(i) no sign bit*
- *(ii) all n bits represent the magnitude of the integer that is $|N|$.*

2^n positive values and zero can be represented. The range of integers is $0, 1, \ldots, 2^n - 1$, that is $0 \leq N < 2^n$ or $|N| < 2^n$.

Fact 9.3 (Multiplication by a power of two)**.** *If n-bit integer N is multiplied by 2^k for some integer $k > 1$, then the result $M = N \times 2^k$ has $(n+k)$ bits. The binary representation of M is N shifted left k bit positions (and filling them with zeroes). In other words, the binary representation of M is the concatenation of the binary representation of N with a bit sequence of k zero bits.*

Example 9.2. *Let $N = 5$ whose binary representation in $n = 3$ is* **101***. The $M = N \times 2^5 = 5 \times 32 = 160$. Its binary representation is the concatenation of N's* **101** *and the five zeroes implied by 2^5 i.e.* **00000***. The result is* **10100000** *as needed. Note that $2^5 = 32$ has binary representation* **100000** *i.e. a one followed by five zeroes.*

Fact 9.4 (Integer division by a power of two)**.** *If n-bit integer N is divided by 2^k for some integer $k > 1$, then the result $M = \lfloor N/2^k \rfloor$ has $(n-k)$ bits. The binary representation of M is the binary representation of N after shifting N to the right k bit positions and discarding the k bits past the righmostbit position, or in other words by isolating the $n-k$ bits of N.*

Example 9.3. *Let $N = 160$ whose binary representation in $n = 8$ is* **10100000***. Then $M = \lfloor N/2^6 \rfloor = \lfloor 160 \times 64 \rfloor = 2$. If N* **10100000** *is shifted right 6 positions* **100000** *gets discarded and we are left with* **10***. Equivalently the $n-k = 8-6 = 2$ leftmost bit position are extracted. In either case we are left with* **10** *which is 2 in radix-10, as needed.*

9.2 Signed Mantissa

Fact 9.5 (**n-bit Signed Mantissa**)**.** *An n-bit integer N in signed mantissa representation has*

- *(i) a sign bit that is its leftmost bit, and*
- *(ii) the remaining $n-1$ bits represent the magnitude of the integer that is $|N|$.*

2^{n-1} positive and as many negative integer numbers can be represented including zeroes (a positive and a negative one). The range of integers is $-2^{n-1}+1,\ldots,-1,-0,+0,+1,\ldots,+2^{n-1}-1$, that is, $-2^{n-1} < N < 2^{n-1}$ or $|N| < 2^{n-1}$.

Example 9.4 (8-bit Signed Mantissa)**.** *In 8-bit signed mantissa, the leftmost bit is the sign and the remaining 7 bits the magnitude of the signed integer. Thus $2^8 = 256$ integer values can be represented, 128 positive and 128 negative. One of those positive and one of those negative values is $+0$ and -0 shown below.*

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	} *Signed Mantissa of positive zero* $+0$

7	6	5	4	3	2	1	0	
1	0	0	0	0	0	0	0	} *Signed Mantissa of negative zero* -0

7	6	5	4	3	2	1	0	
0	0	1	0	1	0	1	1	} *Signed Mantissa of* $+43$

7	6	5	4	3	2	1	0	
1	0	1	0	1	0	1	1	} *Signed Mantissa of* -43

For the $+43$ and -43 represenation, the leftmost of the 8 bits is the sign and varies. The remaining 7 righmost bits is the magnitude: $|-43| = |43| = 43$ and both signed integers have the same magnitude. If we convert the 8-bit sequence from radix-2 to radix-10 we get 43 for $+43$ obviously, but 171 for -43's binary representation. Note that $171 = 128+43$ and 128 accounts for the sign bit contribution.

Thus if N is a positive integer number that is $N > 0$ and such that $N < 2^{n-1}$ the signed mantissa representation of N is the same as the 8-bit unsigned integer binary representation of N: the two representation are identical for positive numbers.

For $-N$, a negative number, the signed mantissa representation of $-N$ is the same as the 8-bit unsigned integer binary representation of $128+N = 2^7+N$.

9.3 One's Complement

Fact 9.6 (**n-bit One's complement**). *An n-bit integer N in one's complement representation has*

- *(i) a sign bit that is its leftmost bit, and*
- *(ii) the remaining $n-1$ bits represent the magnitude of integer $N >= 0$ or its complement otherwise.*

2^{n-1} *positive and as many negative integer numbers can be represented including zeroes (a positive and a negative one). The range of integers is* $-2^{n-1}+1,\ldots,-1,-0,+0,+1,\ldots,+2^{n-1}-1$*, that is,* $-2^{n-1} < N < 2^{n-1}$ *or* $|N| < 2^{n-1}$.

Signed mantissa and one's complement represent differently the negative integers including the negative zero.

Example 9.5 (8-bit One's complement). *In 8-bit one's complement, the leftmost bit is the sign and the remaining 7 bits the magnitude of the signed integer or the complement of the magnitude. By complement we mean flipping ones into zeroes and zeroes into ones. Thus* $2^8 = 256$ *integer values can be represented, 128 positive and 128 negative. One of those positive and one of those negative values is* $+0$ *and* -0 *shown below. The positive zero, as before is represented as* **00000000**. *The negative zero is* `11111111`. *This is because in the bit sequence the sign bit is* 1 *indicating a negative number is represented. In order to retrieve the magnitude of this number, we first extract the 7 righmost bits* `1111111` *and then we flip them and they become* `0000000`. *Thus the negative number represented has magnitude 0 and this is* -0.

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	} *One's complement : positive zero* $+0$
1	1	1	1	1	1	1	1	} *One's complement : negative zero* -0
0	1	1	1	1	1	1	1	} *One's complement :* $+127$
1	0	0	0	0	0	0	0	} *One's complement :* -127
0	0	1	0	1	0	1	1	} *One's complement :* $+43$
1	1	0	1	0	1	0	0	} *One's complement :* -43

9.4 Two's Complement

Fact 9.7 (**n-bit Two's complement**). *An n-bit integer N in two's complement representation has*

- *(i) a sign bit that is its leftmost bit, and*
- *(ii) If $N = 0$ its representations is n zero bits.*
- *(iii) If $2^{n-1} > N > 0$ the binary representation of N is the same as the unsigned (and also the one's complement) representation of N.*
- *(v) If $-2^{n-1} \leq N < 0$ the binary represenation of N is derived by writing down the unsigned bit representation of $|N|$ in n bits, flipping all n bits and adding one to the result.*

$2^{n-1} - 1$ positive and 2^{n-1} negative integer numbers can be represented including one zero (a 0-bit sequence). The range of integers is $-2^{n-1}, \ldots, -1, 0, +1, \ldots, +2^{n-1} - 1$, that is, $-2^{n-1} \leq N < 2^{n-1}$.

Example 9.6 (8-bit Two's complement)**.** *In 8-bit Two's complement, the leftmost bit is the sign and the remaining 7 bits can be used to determine the magnitude of the integer. Thus $2^8 = 256$ integer values can be represented, 127 positive and 128 negative; a zero which is an 8-bit all zero sequence has the same sign bit as the positive numbers. The zero is represented as* **00000000**.

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	} *Two's complement : zero* 0
0	1	1	1	1	1	1	1	} *Two's complement MAXINT:* $+127$
1	0	0	0	0	0	0	0	} *Two's complement MININT:* -128
1	0	0	0	0	0	0	1	} *Two's complement :* -127
0	0	0	0	0	0	0	1	} *Two's complement :* $+1$
1	1	1	1	1	1	1	1	} *Two's complement :* -1
0	0	1	0	1	0	1	1	} *Two's complement :* $+43$
1	1	0	1	0	1	0	1	} *Two's complement :* -43

From radix-10 to two's complement. If we start with a negative integer say -128 we find its two's complement representation as follows. Its magnitude is $|-128| = 128$. We write down the magnitude in 8-bit as `10000000`. We first flip the bits to get `01111111` and then add one to the result to get `10000000`. This is the two's complement of -128, also shown above. For -43 we start with its magnitude $|-43| = 43$ in 8-bit binary i.e. `00101011`. We then flip it to get `11010100` and add one to the result to get `11010101`. The latter's is two's complement of -43.

From two's complement to radix-10. Given the two's complement representation of an integer say in 8-bit we can retrieve the value of the integer as follows. Let the 8-bit two's complement be `11111111`. The leftmost bit is the sign bit and it is one. This means we have a negative integer. We first flip all the bits to get `00000000` and then add one to the result. We get `00000001`. This is the magnitude of the negative integer in unsigned representation, which is one. Thus `11111111` is the binary representation of -1.

For the two's complement bit sequence `10000000` we note that it represents a negative number, after flipping we get `01111111` and adding one we get `10000000`. The latter in unsigned representation is a 128. This means that the original `10000000` is -128.

For the two's complement bit sequence `10000001` we note that it represents a negative number, after flipping we get `01111110` and adding one we get `01111111`. The latter in unsigned representation is a 127. This means that the original `10000001` is -127.

Method. Thus the same method works both ways: for a negative number (either because it has a $-$ in its radix-10 representation or a sign bit of 1 in its two's complement representation) flip and add one to the result.

9.5 Fixed-point real numbers

Fact 9.8 (n-bit fixed-point real numbers)**.** *One easy way to deal with real numbers is to assume that n_i of the n bits represent the integer part of the real number and n_d of the n bits represent the decimal part of it, where $n_i + n_d = n$.*

Example 9.7 (8-bit fixed-point real number)**.**
The 8-bit binary sequence represents a fixed-point real number R with $n_i = n_d = 4$. The decimal point is implied after the first four leftmost bit positions and thus the integer part of R is in binary the four leftmost bits i.e. **0001** *or in radix-10, $R_i = 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1$. For the decimal part we first isolate the bit sequence to the right of decimal point* **1100** *and then convert it to radix-10 according to $R_d = 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} = 0.75$. Thus $R = R_i + R_d = 1.00 + 0.75 = 1.75$.*

7	6	5	4	3	2	1	0	
0	0	0	1	1	1	0	0	} *n-bit fixed point with* $n_i = 4$

If we have $n_i = 5$ and $n_d = 3$, the same bit sequence implies a decimal point after the first five leftmost bit positions and thus the integer part of R is in binary the five leftmost bits i.e. **00011** *or in radix-10, $R_i = 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 3$. For the decimal part we first isolate the bit sequence to the right of decimal point* **100** *and then convert it to radix-10 according to $R_d = 1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} = 0.50$. Thus $R = R_i + R_d = 3.00 + 0.50 = 3.50$.*

7	6	5	4	3	2	1	0	
0	0	0	1	1	1	0	0	} *n-bit fixed point with* $n_i = 5$

9.6 Floating-Point real numbers

Definition 9.1 (Normalized real numbers)**.** *A normalized real number has a bit that is one to the immediate left of the decimal point or has only one bit on the left of the decimal point.*

Fact 9.9 (Division by a power of two)**.** *If n-bit real number N is divided by* 2^k *for some integer* $k > 1$*, then the result* $M = N/2^k$ *has* $(n-k)$ *integer bits and k additional decimal bits. The binary representation of M is the binary representation of N after shifting N to the right k bit positions.*

Example 9.8. *Real number* **100.** *is not normalized (first part of the definition). There is a period to the right of the second zero bit. Because of this, on the left of the decimal point there is a zero. Moreover there are three bits to the left of the decimal point.*

Example 9.9. *Let* $N = 160$ *whose binary representation in* $n = 8$ *is* **10100000**. *Then* $M = N/2^6 = 160 \times 64 = 2.50$. *If N* **10100000** *or* **1010000.** *is shifted right 6 positions and we are left with* **10.10000** *in binary. (The implied decimal points is beteen the second and third leftmost bit, if the real number is viewed as fixed-point.) Viewing the result in fixed point it gives* $1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} = 2.5$ *as needed.*

Example 9.10 (Normalizing a real or integer number)**.** *Real number* **N= 100.** *is not normalized. However* $M = N/2^2$ *is normalized. In other words,* $N = M \times 2^2$. *Given that* $M = 1.00$ *or just* $1.$*, the N can be rewritten as* $N = 1.0 \times 2^2$. *We have normalized N. It consists of an integer part* **1** *that is on the left of the decimal point, a mantissa* **.0** *that is on the right of the decimal point, and an exmponent* 2 *(not the base).*

Example 9.11 (Normalization resolved)**.** *Real number* **N= 101.** *is not normalized according to the refined definition requiring only one bit on the left of the decimal point. However* $M = N/2^2$ *is normalized, or* $N = M \times 2^2$. *Then* $N = 1.01 \times 2^2$. *The integer part is* **1**, *the mantissa is* **01** *and the exponent is* **2**.

Theorem 9.1 (Properties of real numbers and integers)**.** *Let* a, b, c *be integer or real numbers. The following properties are true.*
(The last or is disjunctive, not exclusive.)

$$
\begin{aligned}
& a+b=b+a && \text{(commutative addition)}\\
& (a+b)+c=a+(b+c) && \text{(associative addition)}\\
& a+0=0+a=a && \text{(identity element for addition is zero)}\\
& a+(-a)=(-a)+a=0 && \text{(inverse of every element exists for addition)}\\
& ab=ba && \text{(commutative multiplication)}\\
& (ab)c=a(bc) && \text{(associative multiplication)}\\
& a\cdot 1=1\cdot a=a && \text{(identity element for multiplication is one)}\\
& a(b+c)=ab+ac && \text{(multiplication is distributive over addition)}\\
& ab=0 \iff a=0 \text{ or } b=0 && \text{(integral domain).}
\end{aligned}
$$

Definition 9.2 (IEEE 754-1985 Standard)**.** *Real numbers in floating-point are represented using the IEEE 754-1985 standard. Be reminded that in IEEE 754-1985 neither addition nor multiplication are associative operations. Thus it is possible that* $(a+b)+c \neq a+(b+c)$. *Thus errors can accumulate when we add.*

9.7 IEEE-754: Single Precision

Fact 9.10 (Normalized real numbers: Mantissa, Exponent, Significand)**.** *A (fully) normalized real number R is (or can be converted into) of the binary notation form* $R = \pm 1.xxxx \times 2^{yyyy}$*, where the integer part is one,* `xxxx` *is the* **fraction or mantissa** *and* `yyyy` *is the* **exponent***. The fraction plus one i.e.* `1.xxxx` *is known as the* **significand** *D. The significand by definition is always a small real number between 1 and 2. or multiply values!* **Do not forget that!**

Definition 9.3.

S:1	E:8	Mantissa:23

} *SP: 32-bit*

Fact 9.11 (IEEE-754 Single Precision(SP))**.** *In IEEE-754, single precision floating-point numbers are derived from a normalized input of the form* $R = \pm 1.F \times 2^{E}$*, where the significand (1.X) is always between 1.0 and 2.0. They have three given parts, a sign bit, an exponent and a mantissa also known as fraction, and an implied part known as the bias B.*

- **S** *is the one-bit sign bit that is the leftmost bit with 0 indicating non-negative and 1 indicating negative,*
- **E** *is the 8-bit exponent,*
- **F** *is the 23-bit fraction,*
- **B** *is the bias (and set* $B = 127$*).*

There are two zeroes in the representation. A zero E and F has sign the sign of the sign bit S. Exponents that all-0 and all-1 are reserved. The quadraplet (S,E,F,B) *determines the quintuplet* $(S,E,F,B,D = 1+F)$*. The floating-point number represented by* (S,E,F,B,D) *is*

$$R = (1-2S) \times (1+F) \times 2^{E-B}$$

The relative precision in SP with a 23-bit fraction is roughly 2^{-23}*, thus* $23\log_{10}(2) \approx 6$ *decimal digits of precision.*

Example 9.12 (Smallest SP value)**.** *Smallest* $E = 1$ *and then* $E - B = 1 - 127 = -126$*. The smallest fraction* $F = \mathbf{all-0}$*, and then* $1.F = (1+F) = 1.0$*. The smallest numbers are then* $\pm 1.0 \times 2^{-126}$.

Example 9.13 (Largest SP value)**.** *Largest E in binary is* **1111110** *and thus* $E = 254$*. Then* $E - B = 254 - 127 = 127$*. The largest fraction* $F = \mathbf{all-1}$*, and then* $1.F = (1+F) \approx 2.0$*. The largest numbers are then* $\pm 2.0 \times 2^{127}$.

Example 9.14 (Radix-10 to SP)**.** *Let* $R = -0.875$ *with the fractional part being* **.111***. Then* $R = (-1)^1 \times 1.11 \times 2^{-1}$*. We obviously have* $S = 1$*, the fraction is* **F= 110 ... 0***. We also have* $E = -1 + B = -1 + 127 = 126$*. The exponent E in 8-bit binary is* **E= 01111110**.

```
                                        S    E            F
10111111 01000000 00000000 00000000   = 1 | 01111110 | 1000000 00000000 00000000
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

} $R = -0.75$ *in SP*

9.8 IEEE-754: Double Precision

Definition 9.4.

S:1	*Exponent:11*	*Mantissa:52*

} *DP: 64-bit*

Fact 9.12 (IEEE-754 Single Precision(DP))**.** *In IEEE-754, double precision floating-point numbers are derived from a normalized input of the form $R = \pm 1.X \times 2^Y$, where the significand (1.X) is always between 1.0 and 2.0. They have three given parts, a sign bit, an exponent and a mantissa also known as fraction, and an implied part known as the bias B.*

- **S** *is the one-bit sign bit that is the leftmost bit with 0 indicating non-negative and 1 indicating negative,*
- **E** *is the 11-bit exponent,*
- **F** *is the 52-bit fraction,*
- **B** *is the bias (and set $B = 1023$).*

There are two zeroes in the representation. A zero E and F has sign the sign of the sign bit S. Exponents that all-0 and all-1 are reserved. The quadraplet (S,E,F,B) determines the quintuplet $(S,E,F,B,D = 1+F)$. The floating-point number represented by (S,E,F,B,D) is

$$R = (1-2S) \times (1+F) \times 2^{E-B}$$

The relative precision in SP with a 52-bit fraction is roughly 2^{-52}, thus $52\log_{10}(2) \approx 13$ decimal digits of precision.

Example 9.15 (Smallest DP value)**.** *Smallest $E = 1$ and then $E - B = 1 - 1023 = -1022$. The smallest fraction $F =$ **all** $-$ **0**, and then $1.F = (1+F) = 1.0$. The smallest numbers are then $\pm 1.0 \times 2^{-1022}$.*

Example 9.16 (Largest DP value)**.** *Largest E in binary is* **11111111110** *and thus $E = 2046$. Then $E - B = 2046 - 1023 = 1023$. The largest fraction $F =$ **all** $-$ **1**, and then $1.F = (1+F) \approx 2.0$. The largest numbers are then $\pm 2.0 \times 2^{1023}$.*

Example 9.17 (Radix-10 to DP)**.** *Let $R = -0.875$ with the fractional part being* **.111**. *Then $R = (-1)^1 \times 1.11 \times 2^{-1}$. We obviously have $S = 1$, the fraction is* **F= 110 . . . 0**. *We also have $E = -1 + B = -1 + 1023 = 1022$. The exponent E in 11-bit binary is* **E= 01111111110**.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

} $R = -0.75$ *in SP*

Example 9.18. *What number is the 32-bit real number in IEEE-754* `110000001010....0`*? Since the sign bit is $S = 1$ we know the number is negative. The following 8 bits are the exponent E* `10000001` *i.e. they represent $E+B = 129$. Then the exponent is $E = 129-127 = 2$. The fractional part is F=010...0 and thus $D = 1.010\ldots0$. Converting D into denary we get $d = 1+1/4 = 1.25$. Thus the number reprsented is* $(1-2S)\times 1.25\times 2^2 = -5.0$

Example 9.19 (Patriot missile bug)**.** *Then represent $1/10$ in SP. The one-tenth representation caused problems in the 1991 Patriot missile defense system that failed to intercept a Scud missile in the first Iraq war resulting to 28 fatalities.*

Fact 9.13 (Smallest real greater than one)**.** *The first single precision number greater than 1 is $1+2^{-23}$ in SP. The first double precision number greater than 1 is $1+2^{-52}$ in DP.*

Note 9.1 (Same algebraic expression, two results)**.** *The evaluation of an algebraic expression when commutative, distributive and associative cancellation laws have been applied can yield at most two resulting values; if two values are resulted one must be a NaN. Thus $2/(1+1/x)$ for $x = \infty$ is a 2, but $2x/(x+1)$ is a NaN.*

9.9 IEEE-754: Double Extended Precision

Definition 9.5 (Double Extended Precision)**.**
In Double Extended Precision the exponent E is at least 15-bit, and fraction F is at least 64-bit. At least 10B are used for a `long double`*.*

S:1	*Exponent:15*	*Mantissa:64*

} *Double Extended Precision: 80-bit*

```
         single precision (SP)  32-bit              double precision (DP)  64-bit
         Bias B=127                                 Bias B=1023
         ------------------------------             ------------------------------
         |S|  E 8-bit |  F  23-bit    |             |S|  E 11-bit | F 52-bit     |
                         Reserved Values
        E          F                   s   E   F
        00000000   0..0                0 0..0 0..0   is positive zero  +0.0
        00000000   0..0                1 0..0 0..0   is negative zero  -0.0
        00000000   X..X  NotNormalized  (1-2S) x 0.F x 2**-126
        11111111   0..0                0 1..1 0..0   is positive Infinity
        11111111   0..0                1 1..1 0..0   is negative Infinity
        11111111   X..X                NaN           Not-a-Number (eg 0/Inf, 0/0, Inf/Inf)

Smallest E:   0000 0001  = 1   - B = -126
Smallest F:   0000 ... 0000    implies    Smallest D: 1.0000 ... 0000 = 1.0      [normalized]
Smallest Nmbr= 0 00000001 0....0 = (1-2S) x 1.0 x 2**-126  ~ (1-2S) 1.2e-38      [normalized]

Largest  E:   1111 1110  = 254 - B =  127
Largest  F:   1111 ... 1111    implies    Largest  D: 1.1111 ... 1111  ~ 2.0     [normalized]
Largest  Nmbr= 0 11111110 1....1 = (1-2S) x 2.0 x 2**127   ~ (1-2S) 3.4e38      [normalized]

Smallest E:   0000 0000  reserved to mean 2**-126 for nonzero F
Smallest F:   0000 ... 0001    implies    Smallest D: 0.0000 ... 0001 = 2**-23 [unnormalized]
Smallest Nmbr= 0 00000000 0....1 = (1-2S) x 2**-23 x 2**-126 = 2**-149        [unnormalized]

Largest  E:   0000 0000  reserved to mean 2**-126 for nonzero F
Largest  F:   1111 ... 1111    implies    Largest  D: 0.1111 ... 1111 ~ 1-2**-23  [Unnormalized]
Largest  Nmbr= 0 00000000 1....1 = (1-2S) x 1-2**-23 x 2**-126 ~ 2**-126(1-2**-23)[Unnormalized]
```

10 Computer Architectures: von-Neumann and Harvard

10.1 Von-Neuman model of computation

Fact 10.1 (**Von-Neumann model: Program and Data in Same Memory**). *Under this architectural model, a central processing unit, also known as the CPU, is responsible for computations. A CPU has access to a program that is being executed and the data that it modifies. The program that is being executed and its relevant data both reside in the same memory usually called* **main memory**. *Thus main memory stores both program and data, at every cycle the CPU retrieves from memory either program (in the form of an instruction) or data, performs a computation, and then writes back into memory data that were computed at the CPU by one of its units in a current or prior cycle.*

10.2 Harvard model of computation

Fact 10.2 (**Harvard model: Program and Data in Different Memories**). *An alternative architecture, the so called* **Harvard model of computation** *or architecture as influenced by (or implemented into) the Harvard Mark IV computer for USAF (1952) was also prevalent in the early days of computing. In the Harvard architecture, programs and data are stored separately into two different memories and the CPU maintains distinct access paths to obtain pieces of a program or its associated data. In that model, a concurrent access of a piece of a program and its associated data is possible. This way in one cycle an instruction and its relevant data can both and simultaneously reach the CPU as they utilize different data paths.*

Fact 10.3 (**Hybrid Architectures**). *The concepts of* **pipelining, instruction and data-level caches** *can be considered Harvard-architecture intrusions into von-Neumann models. Most modern microprocessor architectures are using them.*

10.3 CPU, Microprocessor, Chip and Die

Fact 10.4 (**CPU vs Microprocessor**). **CPU** *is an acronym for* **Central Processing Unit**. *Decades ago all the units that formed the CPU required multiple cabinets, rooms or building. When all this functionality was accommodated by a single microchip, it became known as the* **microprocessor**. *The number of transistors in modern processor architectures can range from about a billion to 5 billion or more (Intel Xeon E5, Intel Xeon Phi, Oracle/Sun Sparc M7).*

Fact 10.5 (**Chip vs Die**). *A* **chip** *is the package containing one or more* **dies** *(actual silicon IC) that are mounted and connected on a processor carrier and possibly covered with epoxy inside a plastic or ceramic housing with gold plated connectors. A* **die** *contains or might contain multiple cores, a next level of cache memory adjacent to the cores (eg. L3), graphics, memory, and I/O controllers.*

10.4 More than one

Fact 10.6 (**Multi-core, Many-core, GPU and more**). *In the past 10-15 years uni-processor (aka single core aka unicore) performance has barely improved. The limitations of CPU clock speeds (around 2-3GHz), power consumption, and heating issues have significantly impacted the improvement in performance by just increasing the CPU clock speed. An alternative that has been pursued is the increase of the number of "processors" on a processor die (computer chip). Each such "processor" is called a* **core**. *Thus in order to increase performance, instead or relying to increasing the clock speed of a single processor, we utilize multiple cores that work at the same clock speed (boost speed), or in several instances at a lower (clock) speeds (regular speed). Thus we now have* **multiple-core** *(or* **multi-core***) or* **many-core** *processors.*

Example 10.1 (Dual-core and Quad-core). *Dual-core or Quad-core refer to systems with specifically 2 or 4 cores. The number of cores is usually (2019) less than 30 (eg Intel's generic Xeon processors), with Intel's Xeon Phi reaching 57-72 cores. Intel's Phi processor is attached to the CPU and work in 'parallel' with the CPU or independetly of it. In such a case a many-core system is called a* **coprocessor**.

Fact 10.7 (GPU). *A GPU (Graphics Processing Unit) is used primarily for graphics processing. CUDA (Compute Unified Device Architecture) is an application programming interface (API) and programming model created by NVIDIA (TM). It allows CUDA-enabled GPU units to be used for General Purpose processing, sequential or massively paprallel. Such GPUs are also known as GPGPU (General Purpose GPU) when provided with API (Application Programming Interface) for general purpose work. A GPU processor (GK110) contains a small number (up to 16 or so) of Streaming Multiprocessors (SM, SMX, or SMM). Each streaming multiprocessor has up to 192 32-bit cores supporting single-precision floating-point operations and up to 64 64-bit cores supporting double-precisions operations. Other cores support other operations (eg. transendental functions). Thus the effective "core count" is in the thousands.*

11 Computer Architectures: Memory Hierarchies

Fact 11.1 (**CPU and Main Memory Speed.**). *A CPU rated at 2GHz can execute 2G or 4G operations per second or roughly two-four operations per nanosecond, or roughly one operation every 0.25-0.5ns. A CPU can fetch one word from maim memory ("RAM") every 80-100ns. Thus there is a differential in performance between memory and CPU. To alleviate such problems, multiple memory hierarchies are inserted between the CPU (fast) and Main Memory (slow): the closer the memory to the CPU is the faster it is (low access times) but also the costlier it becomes and the scarcier/less of it also is. A* **cache** *is a very fast memory. Its physical proximity to the CPU (or core) determines its level. Thus we have L1 (closest to the CPU, in fact "inside" the CPU), L2, L3, and L4 caches. Whereas L2 and L3 are "static RAM/ SRAM", L4 can be "dynamic RAM / DRAM" (same composition as the main "RAM" memory) attached to a graphics unit (GPU) on the CPU die (Intel Iris).*

Fact 11.2 (**Level-1 cache.**). *A level-1 cache is traditionally on-die (same chip) within the CPU and exclusive to a core. Otherwise performance may deteriorate if it is shared by multiple cores. It operates at the speed of the CPU (i.e. at ns or less, currently). Level-1 caches are traditionally Harvard-hybrid architectures. There is an instruction (i.e. program) cache, and a separate data-cache. Its size is very limited to few tens of kilobytes per core (eg. 32KiB) and a processor can have separate jevel-1 caches for data and instructions. In Intel architectures there is a separate L1 Data cache (L1D) and a L1 Instruction cache (L1I) each one of them 32KiB for a total of 64KiB. They are implemented using SDRAM (3GHz typical speed) and latency to L1D is 4 cycles in the best of cases (typical 0.5-2ns range for accessing an L1 cache) and 32-64B/cycle can be transferred (for a cumulative bandwidth over all cores as high as 2000GB/s). Note that if L1D data is to be copied to other cores this might take 40-64 cycles.*

Fact 11.3 (**Level-2 cache.**). *Since roughly the early 90s several microprocessors have become available utilizing secondary level-2 caches. In the early years those level-2 caches were available on the motherboard or on a chip next to the CPU core (the microprocessor core along with the level-2 cache memory were sometimes referred to as the microprocessor slot or socket). Several more recent microprocessors have level-2 caches on-die as well. In early designs with no L3 cache, L2 was large in size (several Megabytes) and shared by several cores. L2 caches are usually coherent; changes in one are reflected in the other ones.*

An L2 cache is usually larger than L1 and in recent Intel architectures 256KiB and exclusive to a core. They are referred to as "static RAM". The Its size is small because a larger L3 cache is shared among the cores of a processor. An L2 cache can be inclusive (older Intel architectures such as Intel's Nehalem) or exclusive (AMD Barcelona) or neither inclusive nor exclusive (Intel Haswell). Inclusive means that the same data will be in L1, L2, and L3. Exclusive means that if data is in L2, it can't be in L1 and L3. Then if it is needed in L1, a cache "line" of L1 will be swapped with the cache line of L2 containing it, so that exclusivity can be maintained: this is a disadvantage of exclusive caches. Inclusive caches contain fewer data because of replication. In order to remove a cache line in inclusive caches we need only check the highest level cache (say L3). For exclusive caches all (possibly three) levels need to be checked in turn. Eviction from one requires eviction from the other caches in inclusive caches. In some architectures (Intel Phi), in the absence of an L3 cache, the L2 caches are connected in a ring configuration thus serving the purpose of an L3. The latency of an L2 cache is approximately 12-16 cycles (3-7ns), and up to 64B/cycle can be transferred (for a cumulative bandwidth over all cores as high as 1000-1500GB/s). Note that if L2 data is to be copied to other cores this might take 40-64 cycles.

Fact 11.4 (Level-3 cache.). *Level-3 caches are not unheard of nowadays in multiple-core systems/architectures. They contain data and program and typical sizes are in the 16-32MiB range. They are available on the motherboard or microprocessor socket. They are shared by all cores. In Intel's Haswell architecture, there is 2.5MiB of L3 cache per core (and it is write-back for all three levels and also inclusive). In Intel's Nehalem architecture L3 contained all the data of L1 and L2 (i.e.* $(64+256)*4KiB$ *in L3 are redundantly available in L1 and L2). Thus a cache miss on L3 implies a cache miss on L1 and L2 over all cores! It is also called LLC (Last Level Cache) in the absence of an L4 of course. It is also exclusive or somewhat exclusive cache (AMD Barcelona/Shanghai, Intel Haswell). An L3 is a victim cache. Data evicted from the L1 cache can be spilled over to the L2 cache (victim's cache). Likewise data evicted from L2 can be spilled over to the L3 cache. Thus either L2 or L3 can satisfy an L1 hit (or an access to the main memory is required otherwise). In AMD Barcelona and Shanghai architectures L3 is a victim's cache; if data is evicted from L1 and L2 then and only then will it go to L3. Then L3 behaves as in inclusive cache: if L3 has a copy of the data it means 2 or more cores need it. Otherwise only one core needs the data and L3 might send it to the L1 of the single core that might ask for it and thus L3 has more room for L2 evictions. The latency of an L3 cache varies from 25 to 64 cycles and as much as 128-256cycles depending on whether a datum is shared or not by cores or modified and 16-32B/cycle. The bandwidth of L3 can be as high 250-500GB/s (indicative values).*

Fact 11.5 (Level-4 cache.). *It is application specific, graphics-oriented cache. It is available in some architecture (Intel Haswell) as auxiliary graphics memory on a discrete die. It runs to 128MiB in size, with peak throughput of 108GiB/sec (half of it for read, half for write). It is a victim cache for L3 and not inclusive of the core caches (L1, L2). It has three times the bandwidth of main memory and roughly one tenth its memory consumption. A memory request to L3 is realized in parallel with a request to L4.*

Fact 11.6 (**Main memory.**). *It still remains relatively slow of 60-110ns speed. Latency is 32-128cycles (60-110ns) and bandwidth 20-128GB/s (DDR3 is 32GiB/sec). It is available on the motherboard and in relatively close proximity to the CPU. Typical machines have 4-512GiB of memory nowadays. It is sometimes referred to as "RAM". As noted earlier, random access memory refers to the fact that there is no difference in speed when accessing the first or the billionth byte of this memory. The cost is uniformly the same.*

Definition 11.1 (**Linearity of computer memory**). *Memory is a linear vector. A memory is an* **array of bytes***, i.e. a sequence of bytes. In memory M, the first byte is the one stored at* $M[0]$*, the second one at* $M[1]$ *and so on. A byte is also a sequence of 8 binary digits (bit).*

Big Endian vs Little Endian. If we plan to store the 16-bit (i.e. 2B) integer 0101010111110000 in memory locations 10 and 11, how do we do it? Left-part first or right-part first (in memory location 10)? This is what we call **byte-order** and we have **big-endian** and **little-endian**. The latter is being used by Intel and the formed in powerPC architectures.

```
    BigEndian       LittleEndian(Intel architecture)
10: 01010101        11110000
11: 11110000        01010101
```

Fact 11.7 (**Multi-cores and Memory.**). *To support a multi-core or many-core architecture, traditional L1 and L2 memory hierarchies (aka cache memory) are not enough. They are usually local to a processor or a single core. A higher memory hierarchy is needed to allow cores to share memory "locally". An L3 cache has been available to support multi-core and more recently (around 2015) L4 caches have started appearing in roles similar to L3 but for specific (graphics-related) purposes. When the number of cores increases beyond 20, we talk about* **many-core** *architectures (such as Intel's Phi). Such architectures sacrifice the L3 for more control logic (processors). To allow inter-core communication the L2 caches are linked together to form a sort of shared cache.*

12 Hard-Disk Drives (HDD)

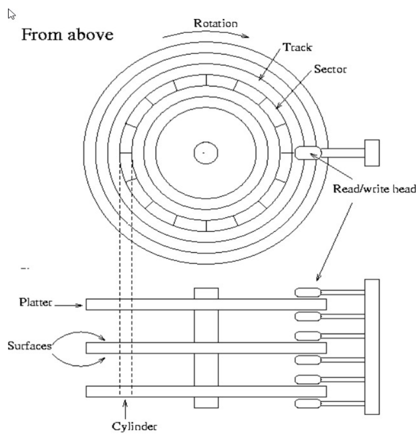


Figure 7: Source: https://www.tldp.org/LDP/sag/html/hard-disk.html

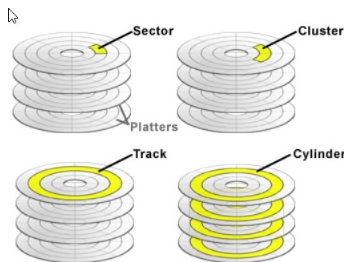


Figure 8: Source: https://superuser.com/questions/974581/chs-to-lba-mapping-disk-storage

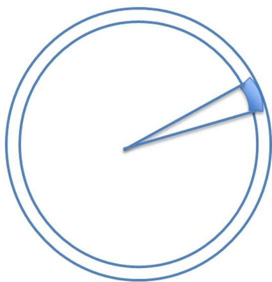


Figure 9: Track and Sector

12.1 Definitions related to HDD

- **Platter (or just disk)** It is ia circular disk. It consists of two surfaces also known as sides: up and down. Both sides (surfaces) can be read/written into. Thus every side of every platter has an associated mechanism known as head to facilitate the reading/writing of information on it. All platters rotate in unison. Usually, one platter or one side of one platter is for control purposes and unused by the user. The remaining ones are utilized for data preservation.
- **Arm and Heads** The Arm contains the disk controller. Attached to the arm are the heads. The number of heads is equal to the number of platters times two. Heads move in unison assisted by the arm. Arms/Head move parallel to surface of platters. If you view a platter as a circular surface the arm and its attached heads moves from the outside periphery to the inside or from the inside to the ouside periphery of a (the) platter(s). Note that only ONE head is active for read and write even though all of them might be over a platter area.
- **Track** It is a concentric circular band (region) on a platter's surface or side. Tracks might be numbered from the outside periphery to the inside or the other way around for ease of reference. The density of tracks is expressed in KTPI (thousands of tracks per inch)
- **Cylinder** All tracks of the same radius from the center of a platter, over all sides of all platters form a cylinder. The number of tracks (over a platter) is thus equal to the number of cylinders (of the HDD).
- **Sector** A sector is a piece of a track at a given arc range. Every track has the same fixed number of sectors as any other track even if tracks on the outside are longer than tracks on othe inside. Thus if tracks have 60 sectors, the first track is between degree 0 and 6, the next one between 6 and 12 and so on. A head reads or writes a sector worth of data.
- **Cluster** A set of consecutive sectors of a track form a cluster.
- **Spindle Speed / Rotation** Platters (disks) rotate very fast. The spindle speed of a drive is the rotational speed of its platters.

12.2 HDD Operation

The hard disk controller receives a request for I/O to be performed on a particular sector number. The data received by the disk controller are then mapped to a platter number, side of a platter (up or down), track within a platter, and sector within a track.

12.3 Seek

The controller moves the arm and its heads horizontally and parallel to the surface of the platters to identify the correct track. There is some initial delay due to controller overhead, then the arm/heads move, and then the arm/heads brake before they settle over a given track. (Think of it as initial delay, acceleration, steady move, and braking and settling.)

Seek Time is the time for arm/heads to move to the right track from their current position. Seek time depends on the initial position (starting track of the heads) and the final/settline position of the heads (destination/target track). This time includes settling time (braking time) and might or might not include controller overhead.

Maximum Seek Time is defined as the time to move the arm/heads to the most inside track from the most outside track or the other way around.

In the 1950s and 1960s maximum seek time was 600ms. In the 1970s it went down to 25ms. First PC-based HDD in the 1980s has maximum seek time around 120ms and nowadays this is around 20-30ms for laptop or desktop drives and 10-12ms for server drives.

The **Average Seek Time** is a better measure of performance. The average seek time is defined as one-third of max seek time. A proof is to be shown later. (Think of it that you figure seek time for every possible initial position and every possible ending position of the heads.) The average seek time for a typical HDD is 8-9ms for a read and 9-10ms for a write operation. Server HDD might have average seek time as low as 4ms.

A **Track-to-Track Seek Time** refers to the time it takes for heads to move minimally by one track. Most of this time is settling time and possibly controller overhead if it is not accounted separately. Typical Track-to-Track seek time is 1 to 1.2ms.

Controller Overhead is less than 2ms for typical drives.

After settling the heads are over the appropriate track. At this point the controller activates one head for the relevant platter and the relevant surface (up or down) involved in the I/O. One and only one head is active in the remainder.

12.4 Rotational Delay or Latency

The active head waits for the appropriate sector to appear under or over the head. (A surface/side can be under a head if it is an up surface; it can be over the head if it is a down surface.) This is because the platters (i.e. disks) rotate at spindle speed also known as rotational speed that varies from 3600RPM to 5400RPM (laptop drives) to 7200RPM (some desktop and regular server drives). The unit RPM refers to Rotations/Revolutions Per Minute.

Rotational delay or Latency Time refers to the time it takes for the appropriate sector be under or over the relevant head positioned under or over the active head. A 7200RPM drive completes one rotation in approximately 8.33ms.

$$\frac{\textbf{Time}}{\textbf{Rotation}} = \frac{\textbf{1mn}}{\textbf{7200R}} = \frac{\textbf{60s}}{\textbf{7200R}} = \frac{\textbf{60,000ms}}{\textbf{7200R}} = 8.33ms/Rotation = 8.33ms/R \quad \textbf{R} = \textbf{Rotation}$$

Because a head might just have missed a specific sector or might just catch a specific sector of a track a more relevant measure of rotational delay is **Average Latency or Average Rotational delay.**

Average Latency Time or Average Rotational Delay is defined to be one-half of the rotational delay. Thus for a 7200RPM disk this is $(1/2) \times 8.33 = 4.17$ms/R.

12.5 Transfer Time

The active head has made contact with the appropriate sector. Data get transferred from the sector (read operation) or transferred into the sector (write operation).

Sector size is 512B. Modern hard disk drives support 4KiB (4096B) sectors. In the latter case the term **logical sector size** is defined as 512B and the term **physical sector size** is defined as 4KiB (4096B).

Transfer data speed for modern HDD is expressed in bytes/s or multiples of bytes/s. Rarely in bits/s. Beware of dubious multiples of bytes such KB and MB and their definitions. Typical data transfer speed rates are in the aread of 200,000KiB/s.

Transfer time is the time it takes for the head to transfer data to/from the disk.

This time is quite straightforward to figure out if the operation involves one sector (of one track of one cylinder of one side of one platter). Multi-sector I/O on different tracks are more complicated to analyze. In most cases when the transfer involves more that sector size worth of data, we ignore additional access, latency costs.

12.6 More on Sectors

A sector of a track stores not only data but also additional information. Some of it relates to the data directly: it is error correcting information in the form of error correcting codes (ECC) that can be used to retrieve or recover information from minor accidents (eg scratches). Additional information is available to prepare the head to read information or synchronize with the sector underneath or over it.

Therefore, a 512B sector is preceded by 15B of gap, sync, and sector address data, followed by 50B of ECC (Error Correcting Code) data (40 10-bit).Therefore a head effectively reads $15 + 512 + 50 = 577$B when it reads a (logical) sector. In other words $512/577 = 88\%$ of the sector data read is sector data for the application.

For a 4096B sector, things change slightly after the sector: the 15B of gap, sync and sector address data still appear before the 4096B sector data. They are followed by 100B of ECC (80 10-bit).

12.7 An Example: HDD around 2019

A modern 7200RPM server hard disk drive with capacity (10TB or 10TiB?) usually has 7 platters (disks) with 14 heads. One of the 14 sides is used for controlling the disk, the remaining 13 sides for data storage. Data density nowadays is approximately 1.5TiB per platter or equivalently 0.75TiB per side. (Logical) sector size is defined as 512B, and thus a (Physical) sector size is defined as 4KiB (4096B) as already mentioned. A physical sector emulates 8 logical sector (8x512=4096) Average seek time is 8-9ms depending on whether a read or write is performed, wih average rotational delay (average latency) being 4.16-4.17ms which is one half of the rotational speed of 8.33ms/R of a 7200RPM HDD. Controller overhead is no more than 2ms. I/O transfer rate is approximately 200,000 KiB/s.

Step 1. The **time to read** one logical sector (512B) is the sum of **disk access** time plus **transfer time**.

Step 2. Disk access time includes **controller overhead**, **average seek time** and **average rotational delay / average latency**. Contoller overhead is about 2ms, average seek time is roughly 8ms, and average rotational dealy is 4.17ms. The total disk access time is 14.17ms.

Step 3. Transfer rate is 200,000KiB/s. Thus the **transfer time** for a 512B sector is negligible at 0.002ms.

Step 4. Thus the **time to read** one logical sector (512B) is 14.17ms.

Effective Transfer Rate is determined by actual byte transferred in the unit of time. If we use the time to read one logical sector, we have 512B transferred in 14.17ms, which gives an effective transfer rate of

$$\frac{512B}{14.17ms} = \frac{512B}{14.17 \times 10^{-3}s} = 36132B/s \approx 35KiB/s$$

13 Average Seek Time vs Maximum Seek Time

Fact 13.1. *Assume a Hard-Disk Drive (HDD) contains $N+1$ tracks indexed 0 through N. The maximum seek time of an arm/heads movement, expressed in number of tracks, is N, when the heads move from track 0 to track N or the other way around. The average seek time, expressed in number of tracks, is approximately $N/3+1/3 \approx N/3$.*

Proof. If the arm/heads move from track i to track j, the distance in track covered is $|i-j|$. Thus the average seek time A, in terms of number of tracks, is the average over all initial and over all final positions of the arm/heads. The number of choices for i is $N+1$ (i.e. 0 through N) and likewise for j. Therefore Therefore

$$A = \frac{\sum_{i=0}^{N}\sum_{j=0}^{N}|i-j|}{(N+1)^2} = \frac{1}{(N+1)^2}\cdot\sum_{i=0}^{N}\sum_{j=0}^{N}|i-j| = \frac{1}{(N+1)^2}\cdot S.$$

We compute next the sum S.

$$
\begin{aligned}
S &= \sum_{i=0}^{N}\sum_{j=0}^{N}|i-j| \\
&= \sum_{i=0}^{N}\left[\sum_{j=0}^{i}|i-j| + \sum_{j=i+1}^{N}|i-j|\right] \\
&= \sum_{i=0}^{N}\left[\sum_{j=0}^{i}(i-j) + \sum_{j=i+1}^{N}(j-i)\right] \\
&= \sum_{i=0}^{N}\left[\sum_{j=0}^{i}i - \sum_{j=0}^{i}j + \sum_{j=i+1}^{N}j - \sum_{j=i+1}^{N}i\right] \\
&= \sum_{i=0}^{N}\left[i(i+1) - i(i+1)/2 + \sum_{j=0}^{N}j - \sum_{j=0}^{i}j - i(N-i)\right] \\
&= \sum_{i=0}^{N}[i(i+1) - i(i+1)/2 + N(N+1)/2 - i(i+1)/2 - i(N-i)] \\
&= \sum_{i=0}^{N}[N(N+1)/2 - i(N-i)] \\
&= \sum_{i=0}^{N}N(N+1)/2 - N\cdot\sum_{i=0}^{N}i + \sum_{i=0}^{N}i^2 \\
&= N(N+1)^2/2 - N^2(N+1)/2 + N(N+1)(2N+1)/6
\end{aligned}
$$

After some minor calculations we obtain the following

$$
\begin{aligned}
S &= \sum_{i=0}^{N}\sum_{j=0}^{N}|i-j| \\
&= \frac{3N(N^2+2N+1) - 3N^3 - 3N^2 + 2N^3 + 3N^2 + N}{6} \\
&= \frac{3N^3 + 6N^2 + 3N - 3N^3 - 3N^2 + 2N^3 + 3N^2 + N}{6} \\
&= \frac{2N^3 + 6N^2 + 4N}{6} = \frac{N^3 + 3N^2 + 2N}{3} \\
&= N(N+1)(N+2)/3.
\end{aligned}
$$

Therefore from Equation 1 by replacing into Equation 1 we obtain the following.

$$
\begin{aligned}
A &= \frac{1}{(N+1)^2}\cdot S = \frac{1}{(N+1)^2}\cdot\frac{N(N+1)(N+2)}{3} = \frac{N}{3} + \frac{N}{3(N+1)} = \frac{N}{3} + \frac{N+1-1}{3(N+1)} \\
&= \frac{N}{3} + \frac{1}{3} - \frac{1}{3(N+1)} \to \frac{N}{3} + \frac{1}{3}.
\end{aligned}
$$

□

14 Constants, Variables, Data-Types

Fact 14.1 (**Constant.**). *If the value of an object can never get modified, then it's called a constant.* 5 *is a constant, its value never changes (ie. a* 5 *will never have a value of* 6*).*

Fact 14.2 (**Variable.**). *In computer programs we also use objects (names, aliases) whose values can change. Those objects are known as a* **variable**.

Fact 14.3 (**Data-type.**). *In a programming language, every variable has a data-type, which is the set of values the variable takes. Moreover, the data-type defines the operations that are allowable on it.*

Example 14.1. *What are data types supported by C, C++, or Java?*

In mathematics, an integer or a natural number is implicitly defined to be of arbitrary precision.

Fact 14.4 (**Built-in or primitive data-types. Composite data-types.**). *Computers and computer languages have* **built-in** *(also called* **primitive***) data-types for integers of finite precision. These primitive integer data-types can represent integers with 8-, 16-, 32- or (in some cases) 64-bits or more. An integer data-type of much higher precision is only available not as a primitive data-type but as a* **composite** *data-type through* **aggregation** *and* **composition** *and built on top of primitive data-types. Thus a composite data-type is built on top of primitive data types.*

Fact 14.5 (Java's integer primitive data types). *Java's (primitive) (signed) integer data types include* **byte**, **short**, **int**, *and* **long**.

- *In java a* **byte** *is an 8-bit signed two'complement integer whose range is* $-2^7 \ldots 2^7 - 1$.
- *In java a* **short** *is a 16-bit signed two'complement integer whose range is* $-2^{15} \ldots 2^{15} - 1$.
- *In java an* **int** *is a 32-bit signed two'complement integer whose range is* $-2^{31} \ldots 2^{31} - 1$.
- *In java a* **long** *is a 64-bit signed two'complement integer whose range is* $-2^{63} \ldots 2^{63} - 1$.

The default value of a variable for **byte**, **short**, **int** *is 0, and for* **long** *it is a 0L.*

Fact 14.6 (Java's other primitive data types). *Java's other data types include* **float**, **double**, **boolean**, *and* **char**.

- *In java a* **float** *is a 32-bit IEEE 754 floating-point number.*
- *In java a* **double** *is a 64-bit IEEE 754 floating-point number.*
- *In java an* **boolean** *has only two possible values:* **true** *and* **false**.
- *In java a* **char** *is a 16-bit Unicode character.*

The default value of a variable for **float**, **double**, **boolean**, *and* **char** *is* `0.0f`, `0.0d`, `false` *and '\u0000' i.e* `U+0000`.

Example 14.2 (Composition : Arrays)**.** *One way to build a composite data-type is through an aggregation called an* **array***: an array is a sequence of objects of (usually) the same data-type. Thus we can view memory as a vector of bytes. But if those bytes are organized in the form of a data-type a sequence of elements of the same data-type becomes known as an array (rather than a plain vector).*

Sometimes the data type of a variable is assigned by default, depending on the value assigned to the variable. The data-type of the right-hand side determines the data-type of x in $x = 10$: in this case it is of "number data type". In some other cases we explicitly define the data type of a variable. A programming language such as C++ consists of primitive data-types such as `int, char, double` and also composite data types that can be built on top of them such as array, struct and class.

Example 14.3. *What are the primitive data types of C, C++, or Java? What are the composite data types of C, C++, or Java?*

Fact 14.7 (What is a Data Model (DM)?). *It is an abstraction that describes how data are represented and used.*

Example 14.4. *What is the data model of C, C++, or Java. Do they differ from each other? The whole set of data types and the mechanisms that allow for the aggregation of them define the data model of each programming language.*

Fact 14.8 (Weakly-typed and strongly-typed languages.). *In a weaky-typed language the data type of a variable can change during the course of a program's execution. In a strongly-type language as soon as the variable is introduced its data-type is explicitly defined, and it cannot change from that point on. For example*

```
Weakly Typed Language such as MATLAB
x=int8(10); % x is integer (data type)
x=10.12;    % x is real number (data type)
x='abcd';   % x is a string of 4 characters (data type)

Strongly Typed Language such as C, C++, or Java
int x   ;   % x is a 32-bit (4B) integer whose data type can not change in the program
x=10;x=2;   % ok
x=10.10 ;   % Error or unexpected behavior: right hand-side is not an integer.
```

Fact 14.9 (Definition vs Declaration.). *In computing we use the term* **definition** *of a variable to signify where space is allocated for it and its data-type explicitly defined for the first time, and* **declaration** *of a variable to signify our intend to use it. A declaration assumes that there is also a definition somewhere else, does not allocate space and serves as a reminder. For a variable there can be only ONE definition but MULTIPLE declarations. This discussion makes sense for compiled languages and thus* `int x` *serves above as a definition of variable* `x`*. For interpreted languages, separate definitions are usually not available and declarations coincide with the use of a variable. Thus we have three declarations that also serve as definitions of x in the weakly-typed example each one changing the data type of x. In the latter example variable x is defined once and used twice (correctly) after that definition.*

Fact 14.10 (What is an abstract data type (ADT)?). *An abstract data type (ADT) is a mathematical model and a collection of operations defined on this model.*

Fact 14.11 (The ADT Dictionary.). *For example a Dictionary is an asbtract data type consisting of a collection of words on which a set of operations are defined such as Insert, Delete, Search.*

Fact 14.12 (What is a data-structure?). *A data structure is a representation of the mathematical model underlying an ADT,* **or,** it is a systematic way of organizing and accessing data.

Fact 14.13 (Does it matter what data structures we use?). *For the Dictionary ADT we might use arrays, sorted arrays, linked lists, binary search trees, balanced binary search trees, or hash tables to represent the mathematical model of the ADT as expressed by its operations. What data structure we use, it matters if* **economy of space** *and* **easiness of programming** *are important. As* **running/execution** *time is paramount in some applications, we would like to access/retrieve/store data as fast as possible. For one or the other among those data structures, one operation is more efficient than the other.*

Fact 14.14 (Mathematical Function: Input and Output Interface.). *When we write a function such as $f(x) = x * x$ in Mathematics we mean that x is the* **unknown or parameter or ideterminate** *of the function. The function is defined in terms of x. The computation performed is x^2 i.e. $x * x$. The value 'returned' or 'computed' is exactly that $x * x$. When we call a function with a specific input argument we write $f(5)$. In this case 5 is the* **input argument** *or just argument. Then the 5 substitutes for x i.e. it becomes the value of parameter x and the function is evaluated with that value of x. The result is a 25 and thus the value of '$f(5)$' becomes '25'. If we write $a = f(5)$, the value of $f(5)$ is also assigned to the value of variable a. Sometimes we call s the* **output argument***, which is provided by the caller of the function to retrieve the value of the function computed.*

Fact 14.15 (Algorithms.). *We call algorithms* **the methods that we use** *to operate on a data structure. An* **algorithm** *is a well-defined sequence of computational steps that performs a task by converting an (or a set of) input value(s) into an (or a set of) output value(s).*

Fact 14.16 (Computational Problem.). *A* **(computational) problem** *defines an input-output relationship. It has an input, and an output and describes how the output can be derived from the input.*

Fact 14.17 (Computational Problems and (their) Algorithms.). *An* **algorithm** *describes a specific procedure for achieving this relationship, i.e. for each problem we may have more than one algorithms.*