NJIT New Jersey's Science & Technology University

A. V. Gerbessiotis | CS345
Aug 31, 2016 | Fall 2016
Computer Science Fundamentals | Handout 2

1 Powers of two or exponentials base two

1.1. Powers of 2. The expression 2^n means the multiplication of n twos. Thus $2^8 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$ 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 (ˆ is the hat/caret symbol usually co-located with the numeric-6 keyboard key).

Expression	Value
2^0	1
2^1	2
2^4	16
2^8	256
2^{10}	1024
2^{16}	65536
2^{20}	1,048,576
2^{24}	16,777,216
2^{30}	1,073,741,824

Figure 1: Some 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

1.2. Properties of powers. Among these

1.2.1 (Multiplication.) $2^m \cdot 2^n = 2^m\ 2^n = 2^{m+n}$. (Dot $\cdot$ optional.)

1.2.2 (Division.) $2^m/2^n = 2^{m-n}$. (The symbol / is the slash symbol)

1.2.3 (Exponentiation.) $(2^m)^n = 2^{m \cdot n}$.

1.3. Approximations for 2^{10} and 2^{20} and 2^{30}. Since $2^{10} = 1024 \approx 1000 = 10^3$, we have that
$2^{20} = 2^{10} \cdot 2^{10} \approx 1000 \cdot 1000 = 1,000,000 = 10^6$.
Likewise, $2^{30} = 2^{10} \cdot 2^{10} \cdot 2^{10} \approx 1000 \cdot 1000 \cdot 1000 = 1,000,000,000 = 10^9$.
Therefore 2^{20} is **roughly one million** (a bit more) and 2^{30} is **roughly one thousand million** (a bit more).
Is 2^{30} always **roughly a billion** ? Think twice and after reading 1.4 below.

1.4 Billions and Trillions: SI notation. Note that in *British English*, *one billion is one million millions* i.e. *a trillion in American English!* And a *trillion in British English* might have 18 zeroes following the one! So we prefer to say 1G for 10^9 and 1T for 10^{12} using the SI (Systeme International d' unites) prefixes (or prefices)!

2 Logarithms base two

2.1. Logarithm base two ($y = \lg(n)$). The logarithm base two of n is denoted by $y = \lg(n)$ or just $y = \lg n$ or sometimes $y = \log_2 n$, and it is the power y that we need to raise integer 2 to get n.

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

We sometimes write $\lg\lg n$ to denote $\lg(\lg(n))$ and the nesting of logarithms can go on.

Expression	**Value**	**Explanation**
$\lg(1)$	0	since $2^0 = 1$
$\lg(2)$	1	since $2^1 = 2$
$\lg(16)$	4	since $2^4 = 16$
$\lg(256)$	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 3: Logarithms: Base two

2.2 The other logarithms: $\log_{10}(x)$ and $\ln(x)$. If one writes $\log n$, then the writing will be ambiguous. Mathematicians or engineers might assume that it is to the base 10, i.e. $\log_{10} n$ or they can confuse it with $\ln n$, that uses as base the base e of the Neperian logarithms, i.e. $\log_e n = \ln n$, where e is the base of the Neperian logarithms $e = 2.7172\ldots$.

$$\ln(x) \quad \textbf{is the base e logarithm i.e.} \quad \log_e(x)$$

$$\log(x) \quad \textbf{MIGHT BE the base 10 logarithm i.e.} \quad \log_{10}(x)$$

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

2.4. Example. Since $2^{10} = 1024$ we have that $\lg(1024)$ is 10.

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

2.5.1 (Product.) $\lg(n \cdot m) = \lg n + \lg m$.

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

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

2.5.4 (Change of base.) $n^{\lg m} = m^{\lg n}$.

2.6. Example. 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 $\lg(2^{30}) = 30$. Similarly to paragraph 1.3 of the previous page, $\lg(1{,}000) \approx 10$, $\lg(1{,}000{,}000) \approx 20$ and $\lg(1{,}000{,}000{,}000) \approx 30$.

2.7 How much is $n^{1/\lg n}$? Let $z = n^{1/\lg n}$. Then by taking logs of both sides (and using 2.5.3) $\lg z = (1/\lg n)\lg n = 1$, we have $\lg z = 1$ which implies $z = 2$.

3 Floors/Ceilings, Sets/Sequences, Conjunction/Disjunctions

3.1 The floor $\lfloor x \rfloor$. The floor function is defined as follows: $\lfloor x \rfloor$ is the largest integer less than or equal to x.

3.1.1 Exercise: The floor of $\lfloor 10.1 \rfloor$ and $\lfloor -10.1 \rfloor$. The $\lfloor 10 \rfloor$ is 10 itself. The $\lfloor 10.1 \rfloor$ is 10 as well. The $\lfloor -10.1 \rfloor$ is -11.

3.2 The ceiling $\lceil x \rceil$. The ceiling function is defined as follows: $\lceil x \rceil$ is the smallest integer greater than or equal to x.

3.2.2 The ceiling of $\lceil 10.1 \rceil$ and $\lceil -10.1 \rceil$. The $\lceil 10 \rceil$ is 10 itself. The $\lceil 10.1 \rceil$ is 11 as well. The $\lceil -10.1 \rceil$ is -10.

3.3 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 $\{10, 30, 20\} = \{10, 20, 30\}$.

3.4 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 $\langle 10, 30, 20\rangle \neq \langle 10, 20, 30\rangle$.

3.5 Sets include unique elements; sequences not necessarily. Note the other distinction between a **set** and a **sequence**. 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.

3.6 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).

3.7 "Forall", "there exists" and "belongs to". The symbols $\forall$, $\exists$, $\in$ mean "forall", "there exists" and "belongs to" respectively.

3.8 Conjunction, Disjunction and Negation. The symbols $\wedge$, $\vee$ and $\neg$ denote Conjunction, Disjunction and Negation respectively. For `true`, we might use `true` or `t` or `1`; likewise `false` , or `f` , or `0` for false. Thus $1 \wedge 0 = 0$ and $1 \vee 0 = 1$. And $x \wedge \neg x = 0$ and $x \vee \neg x = 1$. Sometimes we write $\bar{x}$ for $\neg x$. Alternatively we might write $x \wedge y$ or x `AND` y for conjunction. Alternatively we might write $x \vee y$ or x `OR` y for disjunction. Alternatively we might write `NOT`(x) for negation.

4 The factorial and Stirling's approximation formula

4.1 The factorial $n!$. The factorial $n!$ is defined as follows:

$$n! = 1 \cdot 2 \cdot \ldots \cdot n.$$

By definition $0! = 1$ and $1! = 1$. Thus $2! = 2$, $3! = 6$ and so on.

4.2 An upper bound for the factorial $n! \leq n^n$. A trivial upper bound for the factorial is derived by upper-bounding each one of its terms by n.

$$n! \;=\; 1 \cdot 2 \cdot \ldots \cdot n \leq n \cdot n \cdot \ldots \cdot n = n^n.$$

4.3 A lower bound for the factorial $n! \geq (n/2)^{(n/2)}$. A trivial lower bound for the factorial is derived by lower-bounding half of its lower-order terms by 1, and half of its higher-order terms by $n/2$.

$$\begin{aligned} n! &= 1 \cdot 2 \cdot \ldots \cdot (n/2) \cdot (n/2+1) \cdot \ldots \cdot n \geq 1 \cdot 1 \cdot \ldots \cdot 1 \cdot (n/2) \cdot \ldots \cdot (n/2) \\ &= (n/2)^{(n/2)}. \end{aligned}$$

Of course here we "ignored" differences between odd and even values of n. The analysis above is true and correct for even n. Think of for $n = 6$ and $n/2 = 3$ and $n/2 + 1 = 4$ that

$$6! \;=\; 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \geq 1 \cdot 1 \cdot 1 \cdot 4 \cdot 5 \cdot 6 \geq 1 \cdot 1 \cdot 1 \cdot 3 \cdot 3 \cdot 3 = 3^3 = (6/2)^{(6/2)}$$

Otherwise for an odd n one should write,

$$\begin{aligned} n! &= 1 \cdot 2 \cdot \ldots \cdot \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil \cdot \ldots \cdot n \geq 1 \cdot 1 \cdot \ldots \cdot 1 \cdot \lceil n/2 \rceil \cdot \ldots \cdot \lceil n/2 \rceil \\ &\geq 1 \cdot 1 \cdot \ldots \cdot 1 \cdot (n/2) \cdot \ldots \cdot (n/2) \\ &= (n/2)^{(n/2)}. \end{aligned}$$

Think of for $n = 5$ and $\lfloor 5/2 \rfloor = 2$ and $\lceil 5/2 \rceil = 3$ that

$$5! \;=\; 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \geq 1 \cdot 1 \cdot 3 \cdot 4 \cdot 5 \geq 1 \cdot 1 \cdot 2.5 \cdot 2.5 \cdot 2.5 = 2.5^3 \geq (5/2)^{(5/2)}$$

4.4 An approximation for $n!$. The following is a simplified approximation of Stirling's approximation formula that follows in 4.5.

$$n! \approx \left(\frac{n}{e}\right)^n,$$

4.5 Stirling's approximation formula for $n!$. It applies for $n \geq 10$.

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right),$$

A corrolary is that $\lg(n!)$ is approximately $n \lg n$ or $\Theta(n \lg n)$ after the assymtotic Θ symbol is introduced!

$$\lg(n!) \approx n \lg n - n \lg e = \Theta(n \lg n).$$

4.6 The choose symbol as in *n choose k* or *choose k objects out of n*.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

5 Bits and Bytes

5.1 Bits and bytes! (The capitalization is English grammar imposed and intended as an unintentional joke!)

5.2 Bit and its notation. A **bit** is an ACRONYM derived from `BInary digiT` and is the minimal amount of digital information. A bit can exist in one of two states: `1` , `0` or `High, Low` or `Up, Down` or `True, False` or `T, F`. The correct notation for a bit is a fully spelled lower-case **bit**. If we want to write down in English 9 binary digits we write down `9bit`; a transfer rate can be 9.2bit/s. The notation `9b` should be considered **nonsense**. A lower-case *b* should never denote a bit! Several publications mistakenly do so, however!

5.3 Byte and its notation. 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. 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`. **Memory size** is usually expressed in bytes or its larger multiples. We never talk of 8,000bit memory: 1,000B is the correct way to describe it! To store 9bit in memory we need 2B (the extra byte is for the 9th bit)! Sometimes we write `10byte` or `10Byte` which are acceptable; don't use them and write 10B instead.

5.4 Aggregates of a Byte. A `word` is defined to be 32-bit, i.e. 4B. A `double-word` is thus 8B. The metric system (i.e. SI i.e. International System) notation for Byte multiples follows.

```
1 bit   =  2 states (0 or 1)   [ NEVER use 1b ]     , 1 byte = 1B =  8bit   [AVOID 1Byte or 1byte]
1 short =  1 wide-byte =2B= 16bit                   , 1 word = 4B = 32bit  ,  1 double-word = 8B = 64bit

Symbol   Factor   Short/Long Name        Memory size and reading
    Ki = 2**10    kibi  kilobinary       1KiB= 1kibibyte = 2**10B,
    Mi = 2**20    mebi  megabinary       1MiB= 1mebibyte = 2**20B,
    Gi = 2**30    gibi  gigabinary       1GiB= 1gibibyte = 2**30B,
    Ti = 2**40    tebi  terabinary       1TiB= 1tebibyte = 2**40B,
    Pi = 2**50    pebi  petabinary       1PiB= 1pebibyte = 2**50B,

           AVOID THE FOLLOWING CONFUSING NOTATION
It's confusing to write 1kB for 1024B as 1k is 1000; Is 1MB equal to 1,000,000B, or 1000kB or 1024kB or 1MiB?
It's confusing to write 1KB for 1024B as K is degree Kelvin not the 1024 multiplier!
Microsoft uses 1GB for 1GiB of main memory/RAM but hard-disk companies uses 1GB for 1,000,000,000B.
```

5.6 How many bytes in 1KB or 1MB or 1GB? Use `1KiB, 1MiB, 1GiB` if you are thinking of $2^{10}B, 2^{20}B, 2^{30}B$ respectively for main memory (RAM) size or disk-drive capacity. A `K` is degrees Kelvin so 1KB is not SI compliant; yet it is being used for 1KiB. The original sinner was `1kB` that was used to denote 1024B; Intel is using `1KB` for 1024B! Avoid `1kB`, avoid `1KB` and use instead `1KiB`! For main memory (aka RAM) Intel or Microsoft use 1GB to mean 2^{30}B. Hard-disk drive manufacturers such as Seagate use 1GB to mean $1,000,000,000$B. Yet Microsoft uses 1GB to mean 2^{30}B of a hard-disk drive. Avoid all problems using 1GiB for 2^{30}B.
Final Question: 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 you can use 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.

5.7 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 8bit.

5.8 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.

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

6 Frequency and Time Domain

6.1 Frequency: Hz vs cycles/s. The unit of frequency is `cycles` per `second`. 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.)

6.2 Time in seconds or submultiples or multipes. The unit of time is the second and it is denoted by `1s` or also `1sec`. Submultiples are `1ms`, `1`μ`s`, `1ns`, `1ps` which are $10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}$ respectively and are pronounced millisecond, microsecond, nanosecond, and picosecond respectively. **Note that millisecond has two ells.**

6.3 Frequency (f) and time (t): $f \cdot t = 1$. The relationship between frequency and time is given by the following equality $f \cdot t = 1$. 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 rated at 25MHz, used to execute 25,000,000 instructions per second; one instruction had a period or execution time of roughly $1/25,000,000 = 40ns$. A moderm CPU rated at 2GHz allows instructions to be completed in $1/2,000,000,000 = 0.5ns$. (And note that in 1990s and also in 2010s retrieving one byte of main memory still takes 60-80ns.)

6.4 Nanosecond and Distance (and speed of light). 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

7.1 Number Systems: Radix or Base. When we write number `13`, implicit in this writing is that the number is a denary (base-10) integer, and thus we utilize digits `0-9` to write this integer down. Since we use ten digits, the **base** or **radix** of this number is 10. We may write this explicitly by saying "base-10 integer 13" or "radix-10 integer 13". If every number that we write is in base-10 / radix-10 we usually omit to mention the base/radix explicitly. Sometimes we just say that the number is in "decimal" or it is a "decimal integer" or a "denary integer". The former "decimal" might cause confusion as "decimal" is associated with the decimal digits or decimal point of a real number. For this reason, the unfamiliar "denary" may and should be used instead. A more convenient way to describe the radix is as a subscript to the number itself: write 13_{10}. (In computing base-8 and base-16 utilize the special prefixes $0o$ or $0x$ or $0X$ as shown below, to indicate the radix/base for such integers or numbers.)

	Base or Radix	# digits used	digits	notation
Binary	2	2	0, 1	1001011_2
Octal	8	8	0..7	113_8 or 0o113
Denary (also, Decimal)	10	10	0..9	75_{10} or 75
Hexadecimal	16	16	0..9 a..f 0..9 A..F	0x4b or 0X4B or $4b_{16}$ or $4B_{16}$

7.2 Natural Integer Numbers: Unsigned Integers. A natural (integer) number is a positive integer. Sometimes a zero is included. In computing we call such integers **unsigned integers**, i.e they are non-negative integers (zero or positive).

7.3 Integer Numbers: Signed Integers. In general an integer number can be positive, negative or zero. In computing we call such integers **signed integers**.

7.4 Real Numbers: Floating-point Numbers. A real number (that includes integer digits, possibly a decimal point, and decimal digits) is called in computing a `floating-point` number. Thus 12.1 or 12.10 or $1.21 \cdot 10^1$ represent a real number. In exponential notation $a \times 10^b$ might also be written as aEb or aeb. Thus 5.1×10^3 is $5.1e3$ or $5.1E3$.

7.5 n-bit or n-digit numbers. When we talk about binary numbers we fix and express explicitly the binary digits of the representation: thus n-bit denotes an n-bit or n-digit binary number. An n-digit number can be base-2, base-10, base-8, or base-16. But for an n-bit number the "bit" indicates a binary digit and thus implicit in this description is the fact that the number is a binary number. Binary numbers can get quite lengthy. We can group three bits or four bits and assign a special symbol to the group to generate its octal or hexadecimal representation instead ($2^3 = 8$ and $2^4 = 16$). For the 8-bit integer `11110000` the **right-most** zero is sometimes called the **least-significant bit** or digit and the **left-most** one is sometimes called the **most-significant bit** or digit. This is if all bits of `11110000` represent its magnitude i.e. its value. When signed integers are represented and the left-most bit is the sign the remaining bits is the magnitude i.e. `1110000` and then the left-most bit of the magnitude might be called the most-significant bit. However this is not universally followed/adhered.

7.6 Example.

```
101 is base-10, base-2, base-8, based-16 representation of 5, 101, 65, 257 respectively
41  cannot be in base-2 because it uses digit 4 which is not one of the binary digits
81  cannot be in base-2, nor base-8 because it uses an 8 available only for base-10, base-16
AB  cannot be in base-2,base-8,base-10, becaused digits A,B are only used in hexadecimal.
```

7.7 How many (minimal number of) bits to represent unsigned integer n, where $n > 0$? The number of bits for n is $\lfloor \lg n \rfloor + 1$ or equivalently $\lceil \lg (n+1) \rceil$. 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.

7.8 Table of some integers in binary, octal, hexadecimal and denary.

Binary	Denary	Hexadecimal	Octal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	10
1001	9	9	11
1010	10	A	12
1011	11	B	13
1100	12	C	14
1101	13	D	15
1110	14	E	16
1111	15	F	17
10000	16	10	20
10001	17	11	21

```
Notes
Octal 17 is formally written as      0o17


Hexadecimal 11 is formally written as 0x11
Hexadecimal FF or ff is written 0XFF or 0xff respectively
```

7.9 Example with $b = 2$. Convert $n =$ 5-digit integer 11001_2 into decimal. Thus 11001_2 is 25_{10} i.e. 25.

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

7.10 Example with $b = 8$. Convert $n =$ 5-digit integer 11001_8 into decimal. Thus 11001_8 is 4609_{10} i.e. 4609.

$$\begin{array}{llllll} 8^4 & 8^3 & 8^2 & 8^1 & 8^0 & \\ . & . & . & . & . & \\ 1 & 1 & 0 & 0 & 1 & \\ = & = & = & = & = & \\ 4096+ & 512 & 0+ & 0+ & 1 & = 4609 \end{array}$$

7.11 Example with $b = 2$. Convert $1_2, 10_2, 100_2, 1000_2, \ldots, 1\,\underbrace{0\ldots0}_{\text{x zeroes}}$ into decimal.

1_2 is 1_{10} i.e. $2^0 = 1$.
10_2 is $2^1 = 2$.
100_2 is 2^2 i.e. 4.

1000_2 is 2^3 i.e. 8.

Likewise $1\,\underbrace{0\ldots0}_{\text{x zeroes}}$ is 2^x.

7.12 Example with $b = 2$. Convert $1_2, 11_2, 111_2, 1111_2, \ldots, \underbrace{1\ldots1}_{\text{x ones}}$ into decimal.

1_2 is 1_{10} is $2^1 - 1$ i.e. a 1.
11_2 is $2^2 - 1$ i.e. 3.
111_2 is $2^3 - 1$ i.e. 7.

1111_2 is $2^4 - 1$ i.e. 15.

Likewise $\underbrace{1\ldots1}_{\text{x ones}}$ is $2^x - 1$.

7.13 Example with $b = 2$. How many non-negative integers can be represented with n bits? Answer is 2^n from 0 to $2^n - 1$.

7.14 Example. Convert $n =$ 8-bit binary `11111111` into octal

```
' 11'111'111    : Group                     : Step 1.
'011'111'111    : Add leading zeroes        : Step 2
  3   7   7     : Convert groups into octal : Step 3
   0o377        : Output                    : Step 4
```

7.15 Example. Convert n = 8-bit binary 11111111 into octal

```
'1111'1111      : Group                      : Step 1.
'1111'1111      : Add leading zeroes         : Step 2
   F    F       : Convert groups into octal : Step 3
   0XFF         : Output                     : Step 4
   or
   0xff         : Output                     : Step 4
```

7.16 Example: Denary to Binary (left to right). Convert $x = 77$ into binary. The result is 1001101_2.

```
              K=6   K=5   K=4   K=3   K=2   K=1   K=0
             2**6  2**5  2**4  2**3  2**2  2**1  2**0
X=77
64 <= 77?       1
X=77-64=13
32 <= 13?       1     0
X = 13
16 <= 13?       1     0     0
X=13
8  <= 13?       1     0     0     1
X=13-8=5
4  <= 5 ?       1     0     0     1     1
X=5-4=1
2  <= 1 ?       1     0     0     1     1     0
X=1
1  <= 1 ?       1     0     0     1     1     0     1
X=1-1=0
Result is 1001101
```

7.17 Example: Denary to Binary (right to left). Convert $x = 77$ into binary right to left. The result is 1001101_2.

```
X= 77 is odd -------------------------------------
                                                 |
X=(77-1)/2=38                                    |
X=38 is even ----------------------------------  |
                                              |  |
X=38/2=19                                     |  |
X=19 is odd  ------------------------------   |  |
                                          |   |  |
X=(19-1)/2=9                              |   |  |
X=9 is odd   --------------------------   |   |  |
                                      |   |   |  |
X=(9-1)/2=4                           |   |   |  |
X=4 is even  ----------------------   |   |   |  |
                                  |   |   |   |  |
X=4/2=2                           |   |   |   |  |
X=2 is even  ------------------   |   |   |   |  |
                              |   |   |   |   |  |
X=2/2=1                       |   |   |   |   |  |
X=1 is odd   ---------------  |   |   |   |   |  |
                          |   |   |   |   |   |  |
X=(1-1)/2=0               1   0   0   1   1   0  1
Output  1001101
```

7.18 Range of an 8-bit unsigned and signed integer.

```
8-bit unsigned                     can represent all 2**8 = 256 unsigned integers from 0..255
8-bit signed (two's complement)    can represent the 256 integers -128 .. -1 0 1 .. 127
                                   consisting of 128 negative values, 127 positives values and zero
```

7.19 Signed integers. In order to be able to represent negative integers we need to somehow represent the sign of the number as positive (say +) or negative (say −). We somehow need to incorporate this sign with the binary digits of the number. The simplest way to accomplish this is to borrow one of the bits of a binary number and use the 0 value of it to represent a positive number and the 1 value of it to represent a negative number. That is too easy. The problem that we might face is zero: Will its sign be positive or negative, i.e. a 0 bit or a `1` bit? How we handle zero, is going to affect the discussion that follows. But let us resolve first one more issue. What bit of a bit sequence will denote the sign? The answer to this question is rather easy: the left-most bit. Note that we use the term left-most rather than the dangerous (in this setting) most-significant digit or most-significant bit. In addition, when we deal with binary (positive and negative) integers we also need to be explicit about the number of digits in the representation. The are three major representations of (negative and non-negative) integers.

7.19.2 A Signed mantissa. An n-bit integer in signed mantissa has one sign bit (the leftmost one) and $n-1$ bits for the value (magnitude). Thus integers in the range $-2^{n-1}, \ldots, 2^{n-1}$ can be represented with two zeroes, a negative and a positive one available. **7.19.2.a Signed Mantissa Example.** Thus for 8-bit integers, $+43$ becomes `00101011` and -43 becomes `10101011`. The left-most (cautiously most-significant-bit) is the sign and differs in $+43$ from -43; the remaining 7 bits are the same `0101011` and correspond to natural number 43. Note that one NEEDS TO TELL US THAT `10101011` is in signed-mantissa representation. Otherwise we might think that this a natural (`unsigned`) integer whose value is 171. Note that $128+43=171$. Thus if x is a positive integer number, then we can represent $-x$ in signed-mantissa 8-bit notation by considering $128+x$, and writing down its bits as if it was a natural number rather than the negative number $-x$. Moreover for `10101011` in signed mantissa, we treat it first as if it is a non-negative integer and retrieve its magnitude 171. Given that we know is in signed mantissa and the left most bit that is a one is the sign the negative representated is the negation of $171-128=43$ i.e it represent -43! Note that in signed mantissa there is a positive zero and a negative zero i.e. `00000000` and `10000000`.

7.19.3 One's complement. It is similar to the signed mantissa representation for range and number of zeros. Thus an n-bit one'complement integer can represent all integers in the range $-2^{n-1}, \ldots, 2^{n-1}$ with two zeroes (the positive zero being an all-zero sequence and the negative zero an all-one bit sequence).

7.19.4 Two's complement. An n-bit integer in two's complement has a left-most bit used as a sign bit, and the remaining $n-1$ bits for its value. There is only one zero (all bits are zero) and the integers represented are in the range $-2^{n-1}, \ldots, 2^{n-1}-1$.

7.19.4.1. Two's complement example. In two's complement we have the left-most bit used as a sign bit as well. There is only one zero. The two's complement of a non-negative integer is the integer itself in binary. Thus 37 is represented as an 8-bit two's complement integer by the bit sequence `00100101`. If we plan to represent -38 though, we first flip the bits i.e. we generate `11011010` and add one to the result obtaining `11011011`, which is two's complement for -38. Thus `00000000` corresponds to zero. What do we make out of `11111111`? Definitely it is a negative integer. We reverse the steps of converting a denary integer into two's complement by first subtracting one to get `11111110`. Then we flip the bits to get `00000001`. Thus `11111111` is -1. For the same reason there is no negative zero as well: if we start with zero, flip the bits to get `11111111` and try to add one to it we end up getting a 9-bit `100000000` that cause overflow problems!

7.20 Java's numeric data types. 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 the first three primitive datatypes is 0 and for the latter a 0L.

7.21. Floating Point in Fixed-Point. One easy way to incorporate real numbers is to assume that in fixed-point n-bit representation, some of the bits represent the integer part (say a of them) and the remaining $n-a$ bits the decimal If one does not want to commit that many bits, a floating-point representation is used where the decimal point is not fixed to a digits or $n-a$ decimals.

```
                  Integer  Fixed-Point   Fixed-Point
                           abcd.efgh     abcde.fgh
  00011100          28       1.75          3.50
abcd.efgh = a x 8 + b x 4 + c x 2 + d x 1 + e x 1/2 + f x 1/4 + g x 1/8 + h x 1/16
abcde.fgh = a x16 + b x 8 + c x 4 + d x 2 + e x 1   + f x 1/2 + g x 1/4 + h x 1/8
```

7.22. Floating Point Values. Let us start with some definitions affecting real numbers. A normalized binary number is one that has a 1 just before (on the left of) the decimal point. Thus 100. looks like 4 but it is not normalized. Its normalized form would be 1.0×2^2 resulting by shifting it to the right two positions. Thus a normalized binary number is always of the form $\pm 1.xxxx_2 \times 2^{yyyy}$ where 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. Real number in floating-points are represented using the IEEE 754-1985 standard. Be reminded that in IEEE 754-1985 neither addition nor multiplication are associative any more. Thus it is possible that $(a+b)+c \neq a+(b+c)$. Thus errors can accumulate when we add or multiply values! **Do not forget that!**

```
         single precision (SP)                        double precision (DP)
              32-bit                                     64-bit
         Bias B=127                                   Bias B=1203
         ------------------------------               ------------------------------
         |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 Number = 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  Number = 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 Number = 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  Number = 0 00000000 1....1  = (1-2S) x 1-2**-23 x 2**-126 ~ 2**-126(1-2**-23)  [Unnormalized]
```

7.23 Single Precision(SP). In single precision the sign `S` is one-bit (left-most one), the exponent `E` is 8-bit and the fraction `F` is 23-bit. Viewing E as an unsigned integer and converting it into its value and converting all the 23-bits into the corresponding fraction ala the fixed-point conversion, the floating-point number represented by the bit-sequence $(S, E, F, D = 1+F)$ is $n = (1-2S) \times (1+F) \times 2^{E-B} = (1-2S) \times (1+F) \times 2^{E-127}$. The relative precision is SP given that it uses a 23-bit fraction is roughly 2^{-23}, thus $23 \log_{10} 2 \approx 6$ decimal digits of precision can be expected. There are two zeroes: when all of E, F are the 0-bit sequences the positive and negative zero is determined by the sign bit

7.24 Double Precision(DP). In double precision the sign `S` is one-bit (left-most one), the exponent `E` is 11-bit and the fraction `F` is 52-bit. Viewing E as an unsigned integer and converting it into its value and converting all the 52-bits into the corresponding fraction ala the fixed-point conversion, the floating-point number represented by the bit-sequence (S, E, F) is $n = (1-2S) \times (1+F) \times 2^{E-B} = (1-2S) \times (1+F) \times 2^{E-1203}$. The relative precision is DP given that it uses a 52-bit fraction is roughly 2^{-52}, thus $52 \log_{10} 2 \approx 16$ decimal digits of precision can be expected. Thus the first double precision number greater than 1 is $1 + 2^{-52}$.

7.23.1 Single Precision Example(s). In order to represent -0.75 in single precision we first normalize it i.e. $-0.75 = -1 \times 1.1_2 \times 2^{-1}$. The binary number (in fixed point) 1.1_2 is 1.5 thus this expressions means that

$-0.75 = -1 \times 1.5 \times 2^{-1}$ which is correct. To determine E, we add the B i.e. $E = B + (-1) = 127 - 1 = 126$. Its 8-bit binary representation is `0111 1110`. The fraction part F is 1 or in 23 bits `1000000 00000000 00000000`. Thus the result is

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

What number is `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 $E = 129$. Then the exponent is $e = E - B = 129 - 127 = 2$. 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 $(1 - 2S) \times 1.25 \times 2^2 = -5.0$

Then represent 1/10 i.e. the 0.1_{10} in SP. The one-tenth representation caused problem in the 1991 Patriot missile defense system that failed to intercept a Scud missile in the first Iraq war resulting to 28 fatalities.

7.24 Same algebraic expression, two results. The evaluations 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.

7.25 Double-Extended Precision(DP). In double-extended precision the sign `S` is one-bit (left-most one), the exponent `E` is at least 15-bit and the fraction `F` is at least 64-bit; at least 10 bytes are used for a `long double`.

8 Limits

8.1. Limits in computing: Asymptotics. In computing limits are for $n \to \infty$ i.e. for asymptotically large values of some parameter n. The parameter n is problem size or problem size-related. Thus for a sequence of n keys, n is the parameter that represents the number of elements in the sequence. For an 2d-array (aka matrix) $n \times n$, parameter n represents the number of its rows or columns: in this latter examples the size of the matrix is n^2.

8.2.

$$\lim_{n\to\infty} n = \infty$$
$$\lim_{n\to\infty} n\lg n = \infty$$
$$\lim_{n\to\infty} \lg n = \infty$$
$$\lim_{n\to\infty} 2^n = \infty$$

8.3.

$$\lim_{n\to\infty} (a/b)^n = \begin{cases} 0 & a < b \\ 1 & a == b \\ \infty & a > b \end{cases}$$

8.4.

$$\lim_{n\to\infty} \frac{2^n}{n!} = 0$$

8.5. Example Apply L'Hospital if needed. Forget about conversion (multiplicative) constants; they won't affect a 0 or an ∞ limit. Thus in the third equality do not spend time thinking of whether $1/n$ must be multiplied with $\ln 2$ or divided by $\ln 2$, let alone with $\lg 2$ (which by the way is equal to one)!

$$\lim_{n\to\infty} \frac{n\lg n}{n^2} = \lim_{n\to\infty} \frac{\lg n}{n} = \lim_{n\to\infty} \frac{(\lg n)'}{(n)'} = \lim_{n\to\infty} \frac{(1/n)}{1} = 0$$

BTW, derivatives are in Section 9.

9 Derivatives and Integrals

9.1.

$$(f(x)g(x))' = f'(x)g(x) + f(x)g'(x), \qquad \left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x)g(x) - f(x)g'(x)}{g^2(x)},$$

9.2. For fixed constant $k > 0$.

$$\left(x^k\right)' = k \cdot x^{k-1}, \qquad (e^x)' = e^x, \qquad (a^x)' = a^x \cdot \ln(a), \qquad (\ln(x))' = 1/x,$$

As also discussed in 8.5, ignore the multiplicative constant $\ln(a)$ in all calculations involving the limits 0 and ∞

9.3. For fixed constant $k > 0$, and for n in place of x we shall ignore multiplicative constants in our calculations.

$$\left(n^k\right)' \equiv n^{k-1}, \qquad (2^n)' \equiv 2^n, \qquad (a^n)' \equiv a^n, \qquad (\ln(n))' \equiv 1/n, \qquad (\lg(n))' \equiv 1/n,$$

9.4.

$$\int x^k dx = x^{k+1}/(k+1) + c, \qquad \int (1/x) dx = \ln(|x|) + c, \qquad \int (e^x) dx = e^x + c, \qquad \int (a^x) dx = a^x / \ln(a) + c,$$

10 Sums of sequences: finite and infinite

10.1. Some constants: Neperian base, π, Euler's gamma, Golden ratio phi ϕ and its Fibonacci-related counterpart.

$$e \approx 2.718281, \qquad \pi \approx 3.14159, \qquad \gamma \approx 0.57721, \qquad \phi = \frac{1+\sqrt{5}}{2} \approx 1.61803, \qquad \hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -.61803.$$

10.2. Finite Sum of integers, integer squares and integer cubes. The first is the **arithmetic** series.

$$A_n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \qquad \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}.$$

10.3. For $a \neq 1$ we have the following finite sums. The first one is the **geometric** series.

$$G_n = \sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}, \qquad \sum_{i=0}^{n-1} ia^i = \frac{(n-1)a^{n+1} - na^n + a}{(1-a)^2}, \qquad (1+d)^n = \sum_{i=0}^{\infty} \binom{n}{i} d^i$$

10.4. For $\mid a \mid < 1$ we have the following infinite sums derived from G_n as $n \to \infty$.

$$\sum_{i=0}^{\infty} a^i = 1 + a + a^2 + \ldots + a^i + \ldots = \frac{1}{1-a}, \qquad \sum_{i=1}^{\infty} a^i = \frac{a}{1-a}$$

10.5. For $\mid a \mid < 1$ we have the following infinite sum by taking the first derivative of 10.4 and multiplying by a.

$$\sum_{i=0}^{\infty} ia^i = a + 2a^2 + \ldots + ia^i + \ldots = \frac{a}{(1-a)^2}; \qquad \text{For } a = 1/2, \text{ we have } \sum_{i=0}^{\infty} i(1/2)^i = 2$$

10.6. The first sum is the **Harmonic** series.

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \int (1/x)dx \approx \ln(n) + \gamma, \qquad \sum_{i=1}^{n} H_i = (n+1)H_n - n, \qquad \sum_{i=1}^{n} iH_i = \frac{n(n+1)}{2} H_n - \frac{n(n-1)}{4}.$$

10.7. For $|a| < 1$.

$$e^a = 1 + a + \frac{a^2}{2!} + \ldots + \frac{a^i}{i!} + \ldots = \sum_{i=0}^{\infty} \frac{a^i}{i!}.$$

10.8. For $|a| < 1$.

$$\ln(1+a) = a - \frac{a^2}{2} + \ldots + \frac{(-1)^{i+1}a^i}{i} + \ldots = \sum_{i=1}^{\infty} (-1)^{i+1} \frac{a^i}{i}, \qquad \ln \frac{1}{1-a} = a + \frac{a^2}{2} + \ldots + \frac{a^i}{i} + \ldots = \sum_{i=1}^{\infty} \frac{a^i}{i},$$

10.9. For $|a| < 1$.

$$\frac{1}{(1-a)^{n+1}} = \sum_{i=0}^{\infty} \binom{i+n}{i} a^i, \qquad \frac{1}{\sqrt{1-4a}} = \sum_{i=0}^{\infty} \binom{2i}{i} a^i, \qquad \frac{a}{1-a-a^2} = a + a^2 + 2a^3 + 3a^4 + \ldots = \sum_{i=0}^{\infty} F_i a^i$$

10.10.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \qquad \sum_{k=0}^{n} \binom{n}{k} = 2^n, \qquad \binom{n}{k} = \binom{n}{n-k}, \qquad \binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

10.11.

$$(1+x)^n = \sum_{i=0}^{\infty} \binom{n}{i} x^i$$

10.A.1. Proof for 10.2 A_n. Let $A_n = 1 + 2 + \ldots + (n-1) + n$. Write forwards and backwards A_n and add up the two.

$$\begin{aligned}
A_n &= 1 + 2 + \ldots + (n-1) + n \\
A_n &= n + (n-1) + \ldots + 2 + 1 \quad \text{Add up this and the previous equation} \\
2A_n &= (n+1) + (n+1) + \ldots + (n+1) + (n+1) \\
2A_n &= n(n+1) \\
A_n &= n(n+1)/2
\end{aligned}$$

11 Asymptotic Notation and the Recurrences

Master Method. $T(n) = aT(n/b) + f(n)$, such that $a \geq 1$, $b > 1$ are constant and $f(n)$ is an asymptotically positive function. Then $T(n)$ is bounded as follows..

M1 If $f(n) = O(n^{\lg_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\lg_b a})$.

M2 If $f(n) = \Theta(n^{\lg_b a})$, then $T(n) = \Theta(n^{\lg_b a} \lg n)$.

M3 If $f(n) = \Omega(n^{\lg_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $0 < c < 1$ and for large n, then $T(n) = \Theta(f(n))$.

There is an alternative formulation for Case 2 (aka M2) of the master method.
M2′ If $f(n) = \Theta(n^{\lg_b a}(\lg n)^k)$, for some non-negative constant k, then $T(n) = \Theta(n^{\lg_b a}(\lg n)^{k+1})$.

A1. **Little-oh.** $f(n) = o(g(n))$, iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

A2. **Little-omega.** $f(n) = \omega(g(n))$, iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

A3. If $f(n) = o(g(n))$, then $f(n) = O(g(n))$.

A4. If $f(n) = \omega(g(n))$, then $f(n) = \Omega(g(n))$.

A5. **Theta (version 1).** $f(n) = \Theta(g(n))$, iff $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$.

B1. **Theta (version 2).** $f(n) = \Theta(g(n))$ iff $\exists$ positive constants $c_1, c_2, n_0 : \; 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \;\; \forall\, n \geq n_0$.

B2. **Big-Omega.** $f(n) = \Omega(g(n))$ iff $\exists$ positive constants $c_1, n_0 : \; 0 \leq c_1 g(n) \leq f(n) \;\; \forall\, n \geq n_0$.

B3. **Big-Oh.** $f(n) = O(g(n))$ iff $\exists$ positive constants $c_2, n_0 : \; 0 \leq f(n) \leq c_2 g(n) \;\; \forall\, n \geq n_0$.

12 Induction

12.1. An axiom is a statement (or proposition) accepted to be true without proof.
An axiom forms the basis for logically deducing other statements.

12.2 A proposition is a statement that is either true or false.
We can also define an axiom as,

12.2.a An axiom is a proposition that is assumed or accepted to be true.

12.3 A proof is a verification of a proposition by a sequence of logical deductions derived from a base set of axioms.

12.4. A theorem is a proposition along with a proof of its correctness.
Every proposition is true or false with reference to a space we first define axiomatically and then build on by establishing more theorems.

12.5 Axioms of Arithmetic.

For example, Peano's axioms, on whose inductive proofs are based, define natural (non negative) numbers. The set of natural (or non-negative integers) numbers is also denoted by N.

Axiom 1 (Peano). 0 is a natural number.
Axiom 2 (Peano). If n is a natural number, then its successor $s(n)$ is also a natural number.
(We prefer to write $n+1$ for the successor $s(n)$ of n.)
Induction is a proof technique to prove a proposition $P(n)$ that depends on a natural number n.

12.6 Induction (the Principle).

Let $P(n)$ be a proposition that depends on a natural (integer) number n. If

(Base case): $P(0)$ is true, and
(Inductive Step): $P(n)$ implies $P(n+1)$ for all natural numbers n, then
(Conclusion): $P(k)$ is true for all natural numbers k.

12.6.1 Why does induction work?
The base case establishes $P(0)$. The Inductive step establishes the chain of implications,

$$P(0) \to P(1), \quad P(1) \to P(2), \quad P(2) \to P(3), \quad \ldots \quad , P(n) \to P(n+1),$$

To fire-up the chain reaction, $P(0)$ must be true. This is established by the base case!

12.7 Strong Induction.

(Base case): $P(0)$ is true, and
(Inductive Step): $P(0), P(1), \ldots, P(n)$ jointly imply $P(n+1)$ for all natural numbers n, then
(Conclusion): $P(k)$ is true for all natural numbers k.

12.8 Induction: Base case other than $P(0)$**.** If $P(0)$ cannot be proven true, find the smallest value a, where a is a natural number such that $P(a)$ is true. The conclusion would then be true for all $P(k)$ such that $k \geq a$. For Strong induction the changes for the Inductive step and Conclusion are shown below.
(Inductive Step, Base Case $P(a)$**):** $P(a), P(a+1), \ldots, P(n)$ jointly imply $P(n+1)$ for all natural numbers $n \geq a$.
(Conclusion): $P(k)$ is true for all natural numbers $k \geq a$.

13 Computer Architectures: von-Neumann and Harvard

13.1 Architecture and Model of computation: Von-Neumann model. 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.

13.2 Architecture and Model of computation: Harvard model. 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.

13.3 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.

13.4 CPU vs Microprocessor. CPU is an acronym for Central Processing Unit. Decades ago all the units that formed the CPU requires 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). 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.

13.5 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 instance at a lower (clock) speeds (regular speed).

Thus we now have **multiple-core** (or **multi-core**) or **many-core** processors. Dual-core or Quad-core refer to systems with specifically 2 or 4 cores. The number of cores is usually (2016) less than 30 (eg Intel's Xeon processors). Sometime many-core processors (eg Intel's Phi) are attached to the CPU and work in 'parallel' with the CPU or independetly of it. In such a case a many-core is called a **coprocessor**. 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.

14 Computer Architecture: Memory Hierarchies

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).

14.1 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 2000GiB/s). Note that if L1D data is to be copied to other cores this might take 40-64 cycles.

14.2 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 1000-1500GiB/s). Note that if L2 data is to be copied to other cores this might take 40-64 cycles.

14.3 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. $(64 + 256) * 4$KiB 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 250-500GiB/s (indicative values).

14.4 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.

14.5 Main memory. It still remains relatively slow of 60-110ns speed. Latency is 32-128cycles (60-110ns) and bandwidth 20-128GiB/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.

14.6 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.

15 Computer Architecture: Numbers and Data Types

15.1 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).

15.2 Variable. In computer programs we also use objects (names, aliases) whose values can change. Those objects are known as a **variable**. Every variable has a **data-type**. What is the data-type of a variable? In most programming languages a statement such as the one below, assigns the value of constant 10 (right hand-side of equality) to become the value of variable x on the left-hand side of the equality.

```
x=10
```

15.3. Data-type. In a programming language, every variable has a data-type, which is the set of values the variable takes and the operations defined (or are allowable) on the data-type.

15.4 Built-in or primitive data-types. Composite data-types. In mathematics, an integer or a natural number is implicitly defined to be of arbitrary precision. 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. 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.

15.5 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.
```

15.6 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.

15.7 Data model. A Data model is an abstraction that describes how data are represented and used.

15.8 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.

15.9 ASCII Characters. In the preceding pages we viewed several 8-bit or so sequences in a variety of ways as positive or negative numbers. A 7-bit bit-sequence can also be viewed as an ASCII character code. Thus a single byte (8 bits) can be used to store characters, not numbers (remember that a byte is the minimum storage that can be used to store information). For ASCII characters that extra bit that goes unused is either a 0 or an error-correction parity bit.

15.10 Table of ASCII characters. The table below contains all 128 ASCII character codes from 0 to 127 arranged in 8 rows (0-7 in octal or hexadecimal) and 16 columns (0-F in hexadecimal). The ASCII code for a character 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`. 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).

15.11 Unicode characters. The Unicode character set uses 2 or more bytes to represent one character. The Unicode character for an ASCII character remains the same if one adds extra zeroes. Thus `DEL` which is `0x7F` in ASCII has UNICODE code `0x007F`. We also write this as `U+007F`.

\	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

15.12 Java character primitive data types. In jave a **char** data type is a single 16-bit Unicode character. Its minimum value is '\u0000' (or `U+0000`) and its maximum value '\uFFFF' (or `U+FFFF`).

15.13 UTF-8. In UNICODE only numeric values are assigned to characters. These numeric values are not required to be ordered in a specific sequence of bytes, of the same number or variable number. There are several encoding schemes to represent Unicode. One of the is UTF-8 where characters are encoded using 1 to 6 bytes. In UTF-8 all ASCII characters are encoded within the 7 least significant digits of a byte and its most-significant bit is set to 0. Thus Unicode characters `U+0000` to `U+007F` that is the ASCII characters are encode simply as byte 0x00 to 0x7F. Thus ASCII and UTF-8 encoding look the same. All characters larger than `U+007F` use 2 or more bytes each of which has the most significant bit set to 1. This means that no ASCII byte can appear as part of any other character since only ASCII characters have a 0 in the most signficant bit position. 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`.

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
1110xxxx 10xxxxxx 10xxxxxx		