TABLE OF CONTENTS.

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGES</th>
<th>TOPICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p1</td>
<td>TABLE OF CONTENTS</td>
</tr>
<tr>
<td>1</td>
<td>p2</td>
<td>Powers (Exponentials) base two</td>
</tr>
<tr>
<td>2</td>
<td>p3</td>
<td>Logarithms and logarithms base two</td>
</tr>
<tr>
<td>3</td>
<td>p4</td>
<td>Bits and bytes</td>
</tr>
<tr>
<td>4</td>
<td>p5</td>
<td>Notation</td>
</tr>
<tr>
<td>5</td>
<td>p8</td>
<td>Boolean functions</td>
</tr>
<tr>
<td>6</td>
<td>p9</td>
<td>Axioms, Propositions, Theorems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integers, Mathematical Induction</td>
</tr>
<tr>
<td>7</td>
<td>p18</td>
<td>Properties of Integers</td>
</tr>
<tr>
<td>8</td>
<td>p19</td>
<td>Collections : Sets</td>
</tr>
<tr>
<td>9</td>
<td>p21</td>
<td>Collections : Sequences</td>
</tr>
<tr>
<td>10</td>
<td>p22</td>
<td>Real numbers; Functions (floor, ceiling)</td>
</tr>
<tr>
<td>11</td>
<td>p23</td>
<td>Inequalities</td>
</tr>
<tr>
<td>12</td>
<td>p24</td>
<td>Factorial (Stirling’s approximation formula)</td>
</tr>
<tr>
<td>13</td>
<td>p26</td>
<td>Limits ; Derivatives ; Integrals</td>
</tr>
<tr>
<td>14</td>
<td>p27</td>
<td>Minimum and Maximum of a function</td>
</tr>
<tr>
<td>15</td>
<td>p28</td>
<td>Sums of Sequences</td>
</tr>
<tr>
<td>16</td>
<td>p33</td>
<td>Comparison of functions</td>
</tr>
<tr>
<td>17</td>
<td>p35</td>
<td>Asymptotics</td>
</tr>
<tr>
<td>18</td>
<td>p36</td>
<td>Informal asymptotic comparison of functions</td>
</tr>
<tr>
<td>19</td>
<td>p37</td>
<td>Asymptotic comparison of functions</td>
</tr>
<tr>
<td>20</td>
<td>p38</td>
<td>Asymptotic notation</td>
</tr>
<tr>
<td>21</td>
<td>p41</td>
<td>Recurrences : Master method</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Substitution method,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Iteration or recursion tree method</td>
</tr>
<tr>
<td>22</td>
<td>p48</td>
<td>Frequency and Time Domain</td>
</tr>
<tr>
<td>23</td>
<td>p49</td>
<td>Number Systems: Denary, Binary, Octal, Hexadecimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operating Systems : page numbers and offsets</td>
</tr>
<tr>
<td>24</td>
<td>p58</td>
<td>ASCII, Unicode, UTF-8 and UTF-16</td>
</tr>
<tr>
<td>25</td>
<td>p61</td>
<td>Signed Integers and Floating-point numbers: binary notation</td>
</tr>
<tr>
<td>26</td>
<td>p73</td>
<td>Computer Architecture: von-Neumann and Harvard models</td>
</tr>
<tr>
<td>27</td>
<td>p75</td>
<td>Memory Hierarchies</td>
</tr>
<tr>
<td>28</td>
<td>p78</td>
<td>Hard-Disk Drives (HDD)</td>
</tr>
<tr>
<td>29</td>
<td>p83</td>
<td>Constants, Variables, DataTypes, ADT</td>
</tr>
<tr>
<td>30</td>
<td>p86</td>
<td>Formulae Collection</td>
</tr>
</tbody>
</table>
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 \(2^{\ast n}\) or \(2^n\) for \(2^n\) (\(\ast\) is the hat/caret symbol usually co-located with the numeric-6 keyboard key).

<table>
<thead>
<tr>
<th>Power</th>
<th>Value</th>
<th>Prefix</th>
<th>Name</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2^0)</td>
<td>1</td>
<td>d</td>
<td>deca</td>
<td>(10^1 = 10)</td>
</tr>
<tr>
<td>(2^1)</td>
<td>2</td>
<td>h</td>
<td>hecto</td>
<td>(10^2 = 100)</td>
</tr>
<tr>
<td>(2^4)</td>
<td>16</td>
<td>k</td>
<td>kilo</td>
<td>(10^3 = 1000)</td>
</tr>
<tr>
<td>(2^8)</td>
<td>256</td>
<td>M</td>
<td>mega</td>
<td>(10^6)</td>
</tr>
<tr>
<td>(2^{10})</td>
<td>1024</td>
<td>G</td>
<td>giga</td>
<td>(10^9)</td>
</tr>
<tr>
<td>(2^{16})</td>
<td>65536</td>
<td>T</td>
<td>tera</td>
<td>(10^{12})</td>
</tr>
<tr>
<td>(2^{20})</td>
<td>1048576</td>
<td>P</td>
<td>peta</td>
<td>(10^{15})</td>
</tr>
<tr>
<td>(2^{30})</td>
<td>1073741824</td>
<td>E</td>
<td>exa</td>
<td>(10^{18})</td>
</tr>
</tbody>
</table>

Figure 1: Powers of two

Prefix Name Multiplier

Prefix Name Multiplier

Definition 1.2 (Properties of powers).

- (Multiplication.) \(2^m \cdot 2^n = 2^{m+n}\). (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 \(2^{20} = (2^{10})^2 \approx 1000^2 = 10^6\), and likewise, \(2^{30} = (2^{10})^3 \approx 1000^3 = 10^9\).

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 $n$.

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: $\log^k n = (\lg n)^k$ are equivalent. We sometimes write $\lg \lg n$ to denote $\lg (\lg (n))$ and the nesting can go on. Note that $\log^{(k)} n$ 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 $\log (x)$ 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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lg (n)$</td>
<td>y</td>
<td>since $2^y = 2^{\lg n} = n$ (by definition)</td>
</tr>
<tr>
<td>$\lg (1)$</td>
<td>0</td>
<td>since $2^0 = 1$</td>
</tr>
<tr>
<td>$\lg (2)$</td>
<td>1</td>
<td>since $2^1 = 2$</td>
</tr>
<tr>
<td>$\lg (256)$</td>
<td>8</td>
<td>since $2^8 = 256$</td>
</tr>
<tr>
<td>$\lg (1024)$</td>
<td>10</td>
<td>since $2^{10} = 1024$</td>
</tr>
<tr>
<td>$\lg (1048576)$</td>
<td>20</td>
<td>since $2^{20} = 1048576$</td>
</tr>
<tr>
<td>$\lg (1073741824)$</td>
<td>30</td>
<td>and so on</td>
</tr>
</tbody>
</table>

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.) $n^{\lg_a m} = m^{\lg_a n}$. Moreover $\lg a = \frac{\log_a n}{\log_2 n}$ (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 $\lg (1,000,000,000) \approx 30$.

Exercise 2.1. How much is $n^{1/\lg n}$? Let $z = n^{1/\lg n}$. Then by taking logs of both sides (and using the exponentiation rule for logarithms) $\lg z = (1/\lg n) \lg n = 1$, we have $\lg z = 1$ which implies $z = 2$.

Exercise 2.2. Exponential function: A Lower bound. For all real $x$, $e^x \geq 1 + x$, where $e = 2.7172\ldots$.

Exercise 2.3. Exponential function: An upper bound (and the lower bound above combined). For all $x$ such that $|x| < 1$, we have that $1 + x \leq e^x \leq 1 + x + x^2$. 
3 Bits and bytes

Note 3.1 (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 3.1 (Bit). The word bit is an acronym derived from binary digit 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. The notation 9b should be considered nonsense.

Definition 3.2 (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 3.3 (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 3.4 (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.

![Figure 5: SI aggregates of a byte](image)

![Figure 6: Other aggregates of a byte](image)

Definition 3.5 (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, $10^6$B and $10^9$B respectively).

Exercise 3.1 (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.
4 Notation

Some preliminaries.

**Definition 4.1** (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 4.2** (universal quantifier ∀). The ∀ symbol is also known as the universal quantifier. It reads as for all.

**Definition 4.3** (existential quantifier ∃). The ∃ symbol is also called the existential quantifier. It reads as there exists in singular or in plural as there exist.

**Definition 4.4** (set membership). Symbol ∈ is the belongs to set membership symbol.

**Definition 4.5** (implication). X ⇒ 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 4.6** (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 4.7** (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 operation that involves the application of the sine trigonometric function on operand x.

Thus operator is a symbol, the operation is the mathematical or computing 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 4.8** (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 4.9 (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 corresponding 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 4.10 (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 4.11 (Integer vs Real indeterminate 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 4.12 (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 4.13 (Sums and Sigma notation).** The sum $a_0 + a_1 + \ldots + a_n$ can be represented in compact form as

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

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 4.14 (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 4.15 (Non-negative integer numbers). A non-negative integer number can be positive or zero.

Definition 4.16 (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 4.1. 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 4.17 (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 4.18 (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 4.19 (Exponential notation). A real number can be expressed in exponential notation in the form \( a \times 10^b \) or equivalently as \( aE_b \) or \( aeb \).

Thus 5.1 \times 10^3 is 5.1e3 or 5.1E3.

Definition 4.20 (Magnitude of a number real or integer). The magnitude of an integer or real number is its absolute value.

Example 4.2 (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.
5 Boolean functions

Definition 5.1 (conjunction and disjunction). We read \( \land \) as ‘conjunction’ i.e. ‘AND’ and \( \lor \) as ‘disjunction’ i.e. ‘OR’.

We provide a bit more information about Boolean functions. All operations are defined primarily for Boolean arguments. Note that an operator is a symbol that denotes the corresponding operation. In general + is a symbol that denotes primarily addition of numbers, but by using operator overloading it can denote other operations including a disjunction.

Definition 5.2 (Boolean). Boolean variables take two values true or false. For true, we might use \( \text{true} \) or \( t \) or \( 1 \). Likewise we might use for false, \( \text{false} \), or \( f \), or \( 0 \). Moreover in programming languages any non-zero value evaluates to true and a zero value evaluates to false. (Thus a non-zero value is true or 1 whether it is 1 or some other non-zero value.)

Definition 5.3 (Logical Operation: Conjunction.). The symbol \( \land \) or \( \cdot \) or \& or \. indicates a conjunction of two boolean variables. The result if true if and only if both evaluate to true. We can write \( x \land y \) or \( x \cdot y \) or \( x \& y \) or sometimes AND\((x,y)\) or \( x\ AND\ y \).

Definition 5.4 (Logical Operation: (Inclusive) Disjunction.). The symbol \( \lor \) or \( + \) or \| indicates a disjunction of two boolean variables. The result if true if and only if at least one evaluates to true. We can write \( x \lor y \) or \( x + y \) or \( x\| y \) or sometimes OR\((x,y)\) or \( x\ OR\ y \).

Definition 5.5 (Logical Operation: Negation.). The symbol \( \neg \) or \( \sim \) indicates the negation of a single boolean variable. The result is true or false depending on whether the variable is false or true respectively. We can write \( \bar{x} \) or \( \sim x \) or NOT\((x)\).

Definition 5.6 (Logical Operation: Exclusive Disjunction.). The symbol XOR or \( \oplus \) indicates an exclusive disjunction of two boolean variables. The result if true if and only if exactly one is true and exactly one is false. We can write \( x\ XOR\ y \) or XOR\((x,y)\) or \( x \oplus y \).
6 Axioms, Propositions, Theorems

In computer science we prove statements. Such statements need to be expressed/stated precisely.

**Definition 6.1** (Proposition). A *proposition is a statement that is either true or false.*

An axiom forms the basis for logically deducing other statements.

**Definition 6.2** (Axiom). *An axiom is a statement accepted or assumed to be true without proof. An axiom is thus a proposition.*

The first proposition below is obviously true; the next one is false. (Integers shown in this section are traditional denary integers also known as decimal i.e. base-10 or radix-10 integers.)

**Proposition 6.1.** $1 + 1 = 2$

**Proposition 6.2.** $1 + 1 = 1$

**Proposition 6.3** (Goldbach’s Conjecture). *Every even integer greater than two is the sum of two prime numbers.*

Proposition 6.1 is true and Proposition 6.2 is false. According to Wikipedia, Proposition 6.3 is true for all integers up to about $4 \cdot 10^{18}$; no one knows whether Proposition 6.3 is indeed true as stated.

**Definition 6.3** (Predicate). *A predicate is a proposition whose truth is dependent on the value of one or more variables.*

Proposition 6.4 below is defined in the form of predicate $P(n)$

**Proposition 6.4.** $P(n):$ For all positive integer (also known as a natural number) n, integer $n^2 + 7$ is prime.

Another way to form this predicate is by writing $P(n): \forall n \in N. n^2 + 7$ is prime. Here we use the universal quantifier to state *for all.* Moreover the set of natural numbers (i.e. positive integers) is denoted by $N$.

**Definition 6.4** (Proof). *A proof is a verification of a proposition by a sequence of logical deductions derived from a base set of axioms.*

**Definition 6.5** (Theorem). A *theorem is a proposition along with a proof of its correctness.*

**Definition 6.6** (Lemma, Corollary). A *lemma is a preliminary or simpler theorem useful to proving a proposition that yields a theorem. A corollary is a proposition that follows from a theorem in few and usually simple logical deductions.*

A lemma precedes, and a corollary follows a theorem.

*Proposition 6.4 can be easily shown to be false by providing a counterexample.*
Proof. (That Proposition 6.4 is false.) For \( n = 3 \), we have that \( n^2 + 7 = 3^2 + 7 = 16 \) and one divisor of 16 other than 1 and 16, is 2. Therefore 16 is not a prime number, it is in fact a composite number and therefore this simple counterexample shows that Proposition 6.4 is false because it is not true for \( n = 3 \).

Counterexample. To prove that this proposition is false it suffices to find a single integer of the form \( n^2 + 7 \) that is not a prime number. Thus determining for \( n = 3 \) that \( 3^2 + 7 \) is not a prime number completes the proof that the proposition is FALSE.

**Proposition 6.5.** \( \exists n \in \mathbb{N} \text{ such that } n^2 + 7 \text{ is prime.} \)

In order to prove that Proposition 6.5 is true, we only need prove it for a single value of \( n \). For \( n = 2 \), we can easily establish that \( 2^2 + 7 = 11 \) is a prime number. Proposition 6.5 is not, however, very interesting.

Every proposition is true or false with reference to a space we first define axiomatically and then build on by establishing more theorems.

**Definition 6.7 (Axioms of Arithmetic: Peano’s axioms).** Peano’s axioms, on whose inductive proofs are based, define natural numbers. The set of natural numbers is denoted by \( \mathbb{N} \) or \( \mathbb{N} \).

**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 \( n \).)

Theorems in mathematics are true because the space to which these theorems apply are based on simple axioms that are usually true.

**Theorem 6.1 (Well Ordered Set Principle).** Every non-empty subset of \( \mathbb{N} \) has a minimal element.

**Theorem 6.2 (Mathematical Induction).** Let \( A \subseteq \mathbb{N} \). Let \( 1 \in A \) and whenever \( k \in A \) then \( k + 1 \in A \). This implies \( A = \mathbb{N} \).

**Proof.** Let \( A^C \) be the complement of \( A \) over \( \mathbb{N} \).

\[
A^C = \mathbb{N} \setminus A = \{k \in \mathbb{N} | k \notin A\}.
\]

Suppose \( A \neq \mathbb{N} \). Then \( A^C \) is a non-empty set and by the well ordered set principle it has a minimal element and call that element \( a \). Since \( 1 \in A \) it is obviously \( a \neq 1 \). This mean \( a > 1 \) and thus \( a - 1 \) is positive. Because \( a \) is the minimal element of \( A^C \), \( a - 1 \) belongs to \( A \). From the mathematical induction principle set \( k = a - 1 \). \( k \in A \) implies \( k + 1 = a \in A \). This contradicts the membership of \( a \) in the complement of \( A \)! Thus \( A^C \) cannot have a minimal element i.e. it must be empty. This implies that \( A = \mathbb{N} \).

In inductive proofs we try to prove that the set of integers \( A \) that satisfy a given property or proposition is all of \( \mathbb{N} \) i.e. \( A = \mathbb{N} \).

**Theorem 6.3.** If \( a, b \in \mathbb{N} \) then \( ab \geq a \). Equality is applicable if and only if \( b = 1 \).
Proof. Use induction on $b$. Let $a \in \mathbb{N}$.

**STEP 1: Base case.** is $b = 1$. Clearly $a \cdot 1 \geq a$ as the former is $a$ by way of $a \cdot 1 = a$; moreover $a \cdot 1 = a$.

**Auxiliary step.** For the inductive step, since $a \geq 1$, we have $a > 0$ and thus $2a = a + a > a + 0$, i.e. $2a > a$.

**STEP 2: Inductive hypothesis.** Suppose that $ab > a$ for some $b \in \mathbb{N}$.

**STEP 3: Inductive step.** We then have to show that $a(b+1) > a$ to prove the inductive step. This is 

$$a(b+1) = ab + a > a + a > a$$

The first $>$ is by the inductive hypothesis; the second $>$ is by the auxiliary step where $a > 0$ implies $2a > a$.

**STEP 4: Conclusion.** It follows that $ab > a$ for all $b \geq 2$. (And of course $ab = a$ for $b = 1$ thus concluding the theorem.)

\[ \square \]

**Theorem 6.4 (Strong Induction).** Let $A \subseteq \mathbb{N}$. Let $1 \in A$ and whenever $\{1, 2, \ldots, k\} \subseteq A$ then $k + 1 \in A$. This implies $A = \mathbb{N}$.

**Strong vs Weak form of induction.** Given that Theorem 6.4 establishes strong induction, we sometimes refer to Theorem 6.2 as Weak Induction. Thus Induction or Mathematical Induction or Weak Induction or also Ordinary Induction are all synonymous and refer to Theorem 6.2. There is one and only one name for Strong Induction of Theorem 6.4.

**Induction as a proof method.** In inductive proofs we try to prove that the set of integers $A$ that satisfy a given property or proposition is all of $\mathbb{N}$ i.e. $A = \mathbb{N}$. In the remainder we ignore properties and focus on propositions. Of particular interest are proposition that depends on an integer variable $n$ and thus we would establish that the range $A$ of $n$ is indeed $\mathbb{N}$ i.e. $A = \mathbb{N}$. Note that we will use the term "integer variable" as a misnomer for natural number. A natural number of always a positive integer, whereas our "integer variable" would allow the possibility of including 0 (but not negative numbers). Thus "integer variable" is an alias for "non-negative integer", "natural numbers", or "natural numbers including 0".

To wrap up all these assumptions let $P(n)$ be a generic proposition that depends on integer variable $n$. Whether we call the indeterminate $k$ or $n$ or $p$ does not matter. What it matters is its range of values which is usually all of $\mathbb{N}$ (i.e. $\mathbb{N}$).
Note 6.1. (Mathematical Induction: the principle). Let $P(n)$ be a proposition that depends on a natural (integer) number $n$. Weak induction is structured as follows.

(Base case): $P(0)$ is true, and

(Inductive Step): $P(n)$ implies $P(n+1)$ for all natural numbers $n$, then

(Conclusion): $P(n)$ is true for all natural numbers $n$, including 0 (base case value).

Note 6.2. (Why does induction work?) The base case establishes $P(0)$, i.e. the truthness of $P(n)$ for the base case value: the base case value is the smallest value for which $P(.)$ can be proven true. For this generic description 0 is the base case value and thus for the base case we show $P(0)$ is true. (Repetition was intentional in the last two sentences.) The Inductive step establishes the chain of implications,

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

To fire-up the chain reaction of implications, we start with the left-hand side of the first implication that contains $P(0)$. The base case established the truthness of $P(0)$. The first implication then established the truthness of $P(1)$, the second of $P(2)$ and so on! Collectively we say that $P(n)$ is true for all $n \geq 0$. This is equivalent to saying $0 \in A$, $1 \in A$, and so using the terminology of Theorem 6.2.

Note 6.3. (Strong Induction) Let $P(n)$ be a proposition that depends on a natural (integer) number $n$. Strong induction is structured as follows.

(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): $P(n)$ is true for all natural numbers $n$, including 0.

Note 6.4. (A 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.

(Base case): $P(a)$ is true, and

(Inductive Step): $P(a), P(a+1), \ldots, P(n)$ jointly imply $P(n+1)$ for all natural numbers $n \geq a$, then

(Conclusion): $P(k)$ is true for all natural numbers $k \geq a$.  

Let \( S_k(n) = 1^k + 2^k + \ldots + n^k \) for any integer \( k > 0 \). We also write \( S_k(n) = \sum_{i=1}^{n} i^k \). The latter term \( \sum_{i=1}^{n} i^k \) indicates a sum that runs from \( i = 1 \) through \( (\text{inclusive}) i = n \). The \( i \)-th term of the sum is \( i^k \).

We shall show that \( S_1(n) = 1 + \ldots + n \) is equal to \( n(n+1)/2 \). We sometimes denote \( S_1(n) \) as \( A(n) \) for the sum of the terms of the arithmetic sequence.

**Example 6.1.** (Arithmetic sequence sum.)

**Proposition 6.6.** For all natural number \( n \geq 1 \), we have that \( S_1(n) = 1 + 2 + \ldots + n = n(n+1)/2 \).

The first step in induction is to identify in a proposition a predicate that depends on a natural-valued variable and also that same natural-valued variable.

**Proof.** Let us call \( P(n) \) the predicate \( S_1(n) = n(n+1)/2 \). We are going to show that \( P(n) \) is true for all (integer \( n \geq 1 \)). That is we shall show that \( S_1(n) = n(n+1)/2 \). The proof is by induction.

1. **Base case: Show that \( P(1) \) is true.** Thus we shall show that \( S_1(n)|_{n=1} = n(n+1)/2|_{n=1} \). The left hand side sum of \( P(1) \) is \( S_1(1) \) i.e. the sum of one term (and that term is 1), which is 1. The right hand side of \( P(1) \) is \((1+1)/2\) which is also 1. Therefore \( P(1) \) is true since the left and right hand sides of \( = \) are equal to one and equal to each other obviously.

2. **Inductive Step: \( \forall n \in \mathbb{N} P(n) \Rightarrow P(n+1) \).** We need to establish the implication i.e. \( P(n) \Rightarrow P(n+1) \) for all \( n \geq 1 \), i.e. we need to show that \( P(n) \) implies \( P(n+1) \). We move from the left hand of the implication to the right hand side.

2.a **Induction hypothesis:** \( P(n) \). \( P(n) \) true, implies \( S_1(n) = 1 + 2 + \ldots + n = \sum_{i=1}^{n} i = n(n+1)/2 \).

2.b **Is \( P(n+1) \) implied?** To show \( P(n+1) \) we need to show that \( S_1(n+1) = 1 + 2 + \ldots + (n+1) = \sum_{i=1}^{n+1} i = (n+1)(n+2)/2 \).

Show 2.b using 2.a. We start from the left-hand side of the latter equality to derive the right-hand side utilizing the induction hypothesis, i.e. the assumption that \( P(n) \) is true which is equivalent to \( S_1(n) = n(n+1)/2 \): we use to derive the third equality from the second one.

\[
1 + 2 + \ldots + (n+1) = \sum_{i=1}^{n+1} i = \left( \sum_{i=1}^{n} i \right) + (n+1) = n(n+1)/2 + (n+1) = n(n+1)/2 + 2(n+1)/2 = (n+1)(n+2)/2
\]

This completes the induction. We proved two things

- We first proved that \( P(1) \) is true.
- and then showed \( P(n) \Rightarrow P(n+1) \) for all \( n \geq 1 \).

\( \square \)

An alternative would have been to show (substituting \( n-1 \) for \( n \))

- that \( P(1) \) is true,
- and then show that \( P(n-1) \Rightarrow P(n) \) for all \( n \geq 2 \).
**Example 6.2.** *(Geometric sequence sum.)* The sum below is known as the geometric series or sum of the geometric sequence. The i-th term of the sum is $x^i$. (We use $m$ to deviate from the standard notation.)

**Theorem 6.5.** Show that for all integer $m \geq 0$, and for any $x \neq 1$,

$$
\sum_{i=0}^{m} x^i = 1 + x + \ldots + x^m = \frac{x^{m+1} - 1}{x - 1}.
$$

**Proof.** We prove the theorem by induction. The natural variable in the theorem is $m$. The predicate $P(m)$ in the theorem that depends on $m$ is

$$P(m): \quad 1 + x + \ldots + x^m = \frac{x^{m+1} - 1}{x - 1}, \text{ for any } x \neq 1.$$

An equivalent way is to write

$$P(m): \quad \sum_{i=0}^{m} x^i = \frac{x^{m+1} - 1}{x - 1}, \text{ for any } x \neq 1.$$

1. **Base case:** $P(1)$ is true. We can show either $P(0)$ or $P(1)$. To show $P(1)$ we need to show that $1 + x$ is equal to $\frac{x^2 - 1}{x - 1}$ which is obviously true as the latter is a stealthy way of writing $(x^2 - 1)/(x - 1)$ which is indeed $x + 1$ as long $x \neq 1$ which is so anyway.

2. **Inductive Step:** $P(m) \Rightarrow P(m + 1)$. Let us show first what $P(m)$ is/implies, i.e. the induction hypothesis. It is equivalent to

$$P(m): \quad 1 + x + \ldots + x^m = \frac{x^{m+1} - 1}{x - 1}.$$

Then $P(m + 1)$ is equivalent to

$$P(m + 1): \quad 1 + x + \ldots + x^{m+1} = \frac{x^{m+2} - 1}{x - 1}.$$

Similarly to the previous example 6.1 we start from the latter’s left-hand side to conclude its right-hand-side by using the former result for $P(m)$. Between the second and third inequality we use the induction hypothesis for $P(m)$ above.

$$1 + x + \ldots + x^{m+1} = 1 + x + \ldots + x^m + x^{m+1} = (1 + x + \ldots + x^m) + x^{m+1} = \frac{x^{m+1} - 1}{x - 1} + x^{m+1}$$

$$= \frac{(x^{m+1} - 1) + x^{m+1}(x - 1)}{x - 1} = \frac{x^{m+2} - 1}{x - 1},$$

which proves that $P(m + 1)$ is true. Base case and inductive step conclude the induction. $\square$

Just because we set up the induction description by using $n$ to represent the natural number, it doesn’t mean that we should always use $n$ or have $n$ in a Proposition. In this example $m$ was the natural number, and the proposition dependend on $m$ (i.e. we had $P(m)$). Moreover the base case use was $P(1)$; an alternative could have been $P(0)$. 
A recursive function is a function that invokes itself. In direct recursion a recursive function \( f \) invokes directly itself, whereas in indirect recursion function \( f \) invokes function \( g \) that invokes \( f \). A quite well-known recursive function from discrete mathematics is the Fibonacci function \( F_n \) or more widely known as the Fibonacci Sequence \( F_n \). The name sequence implies that it includes all the terms \( F_0, F_1, \ldots, F_{n-1}, F_n, \ldots \). The \( n \) indexed term is given by the following recursive formulation.

\[
F_n = F_{n-1} + F_{n-2} \quad \text{if} \quad n > 1
\]

where

\[
F_0 = 0 \quad \text{and} \quad F_1 = 1
\]

Example 6.3 ((Strong induction example).)

Proposition 6.7 (Fibonacci sequence). For any \( n \geq 0 \), we have that \( F_n \leq 2^n \).

Proof. Proof is by strong induction. Let the predicate \( P(n) \) be \( F_n \leq 2^n \). Given that we have \( F_0 \) we shall use a base case \( n = 0 \). Otherwise, if we use a \( n = 1 \) base case, we could prove a lesser result that for \( n \geq 1 \) \( P(n) \) is true.

1. Base case: Show \( P(0) \) is true. To show \( P(0) \) we need to show that \( F_0 \leq 2^0 \). Since \( F_0 \) is 0 and \( 2^0 = 1 \) we have obviously \( F_0 \leq 1 = 2^0 \), and \( P(0) \) follows directly.

2. Inductive Step. Show that \( P(0) \land \ldots \land P(n-1) \Rightarrow P(n) \). The inductive step is of the (strong induction) type “0, 1, \ldots, n − 1, induces n” and not of the type “1, 2, \ldots, n induces n + 1”.

2.a Induction hypothesis: \( P(0) \land \ldots \land P(n-1) \) which is equivalent to \( \forall i, 0 \leq i < n : P(i) \).

We need to show that \( \forall i, 0 \leq i < n : P(i) \Rightarrow P(n) \). \( P(i) \) is equivalent to \( F_i \leq 2^i \). Among the \( i \) less than \( n \) two are of use: \( P(n-1) \) and \( P(n-2) \) that relate to \( F_{n-1} \) and \( F_{n-2} \) respectively.

\[
P(n-2) : \quad F_{n-2} \leq 2^{n-2} \quad \text{and} \quad P(n-1) : \quad F_{n-1} \leq 2^{n-1}
\]

We complete the inductive step by starting with the recurrence and then claiming \( P(n-1) \) and \( P(n-2) \) for \( F_{n-1} \) and \( F_{n-2} \) in its right hand side; this involves turning an equal into less than or equal.

\[
F_n = F_{n-1} + F_{n-2} \leq 2^{n-1} + 2^{n-2} \leq 2^n - 1 + 2^{n-2} = 2 \cdot 2^{n-1} = 2^n
\]

This completes the strong induction.
Exercise 6.1. (Practice Makes Perfect). Do for practice the following examples. Show that for any \( n \geq 0 \)
\[
\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}
\]

Exercise 6.2. (Practice Makes Perfect). Show that for any \( n > 1 \), \( n^2 - 1 > 0 \).

Exercise 6.3. (Practice Makes Perfect). Show that for any \( n \geq 2 \), \( \sum_{i=1}^{n} i \leq \frac{3n^2}{4} \).

Exercise 6.4. (Practice Makes Perfect). Show that for any \( x \geq 3 \), \( \sum_{i=0}^{n-1} x^i \leq \frac{x^n}{2} \).

Exercise 6.5. (Practice Makes Perfect). Show that for any \( n \geq 1 \), \( \sum_{i=0}^{n} i^2 \leq \frac{(n^3 + 2n^2)}{3} \).

Exercise 6.6. (Practice Makes Perfect). What is wrong with the proof of the theorem below? Explain.

Theorem 6.6. All horses of the world are of the same color.

Proof. The proof is (supposed to be) by induction on the number of horses \( n \).

\( P(n): \) In any set of \( n \geq 1 \) horses, all the horses of the set are of the same color.

1. Base Case: Show \( P(1) \) is true. \( P(1) \) is always true as in a set consisting of a single horse, all the horses (there is only one) of the set have the same color.

2. Inductive step : \( \forall n \in N, \text{i.e. } n \geq 1, P(n) \Rightarrow P(n+1) \).

Let us assume (induction hypothesis) that for any \( n \geq 1 \), \( P(n) \) is true. Since we assume \( P(n) \) to be true, every set of \( n \) horses have the same color. Then we will prove that \( P(n+1) \) is also true (inductive step), i.e. we will show that in every set of \( n+1 \) horses, all of them are of the same color. To show the inductive step, i.e. that \( P(n+1) \) is true let us consider ANY set of \( n+1 \) horses \( H_1, H_2, \ldots, H_n, H_{n+1} \). The set of horses \( H_1, H_2, \ldots, H_n \), consists of \( n \) horses, and by the induction hypothesis any set of \( n \) horses are of the same color. Therefore \( \text{color}(H_1) = \text{color}(H_2) = \ldots = \text{color}(H_n) \). The set of horses \( H_2, H_3, \ldots, H_{n+1} \), consists of \( n \) horses, and by the induction hypothesis any set of \( n \) horses are of the same color. Therefore \( \text{color}(H_2) = \text{color}(H_3) = \ldots = \text{color}(H_{n+1}) \). Since from the first set of horses \( \text{color}(H_2) = \text{color}(H_n) \), and from the second set \( \text{color}(H_2) = \text{color}(H_{n+1}) \), we conclude that the color of horse \( H_{n+1} \) is that of horse \( H_2 \), and since all horses \( H_1, H_2, \ldots, H_n \) are of the same color, then all horses \( H_1, H_2, \ldots, H_n, H_{n+1} \) have the same color. This proves the inductive step. The induction is complete and we have thus proved that for any \( n \), in any set of \( n \) horses all horses (in that set) are of the same color.

(Hint: The key to this proof is the existence of horse \( H_2 \). More details on a later page.)
For the Fibonacci sequence we have shown that $F_n \leq 2^n$ for all $n \geq 0$. We can do a little better with the upper bound on $F_n$.

The method of solving recurrences through induction and in particular strong induction is known as the substitution or the guess-and-check method. In order to use it we need to know what to show, i.e. the predicate $P(n)$ must be given to us in advance, eg $F_n \leq 2^n$.

**Exercise 6.7.** Show that for any $n \geq 0$ $F_n \leq 2^{n-1}$.

**Exercise 6.8.** Show that for any $n \geq 1$ $F_n \geq 2^{(n-1)/2}$.

**Exercise 6.9.** (On horses, cows, and tricky inductive arguments). Horses revisited i.e. why the Theorem of Example 6.6 is false. On the previous page we proved that all horses have the same color, an otherwise nonsense statement (or false proposition). What’s wrong with the inductive proof?

Many may argue that the error is in the logic of the inductive step.

The logic is fine, the quantification “for all $n$” is not, since the assumption of always having horse $H_2$ might not be true. The crux of the inductive step is the existence of three ‘different’ horses $H_1, H_2, H_{n+1}$. We first form a set of $n$ horses $H_1, H_2, \ldots, H_n$ and apply the induction hypothesis and then form another set of $n$ horses, $H_2, \ldots, H_n, H_{n+1}$, and apply the induction hypothesis again. Crucial to the proof is that $c(H_1) = c(H_2)$ from the first application of the induction hypothesis, and $c(H_2) = c(H_{n+1})$ from the second thus concluding that $c(H_1) = c(H_2) = \ldots = c(H_n)$.

Let’s see what happens for $n = 1$, i.e. let’s try to show that $P(1) \Rightarrow P(2)$, i.e. show the inductive step for a certain value of $n$ equal to 1. If we try to form $H_1, \ldots, H_n$ this set contains only one ‘horse’ element for $n = 1$: $H_1$. If we try to form $H_2, \ldots, H_{n+1}$ this set contains only one ‘horse’ element $H_2$. There is no common third ‘horse’ $H_k$ in the set containing $H_1$ nor in the set containing $H_2$. This is because the argument in the previous paragraph works only for $n \geq 2$. In that case one set is formed from $H_1, H_2$ and the other set from $H_2, H_3$. However even if we can prove the inductive step nicely in that case there is no way to prove the base case set for $n = 2$ i.e. $P(2)$!

Therefore the inductive step that “we proved” before $P(n) \Rightarrow P(n+1)$ is not true for all $n \geq 1$ but only for $n \geq 2$. This however can not establish the trueness of $P(n)$ for all $n \geq 1$ because $P(2)$ may or may not be true.

What is $P(2)$?

$P(2)$ is “in any set of two horses, both horses are of the same color”.

In conclusion, the whole “horsy argument” breaks down because

A2. $P(n) \Rightarrow P(n+1)$ for all $n \geq 1$,
was not shown, for all $n \geq 1$; it was only proved for all $n \geq 2$, i.e.

A2. $P(n) \Rightarrow P(n+1)$ for all $n \geq 2$,

and thus the base case “$P(1)$ is true” can not be used with the latter version of the inductive step: for the latter we need $P(2)$ to be true WHICH IS NOT!
7 Integers and their properties

Set $\mathbb{Z}$: The set of integers is denoted by $\mathbb{Z}$.
Set $\mathbb{Z}^*$: The set of non-zero integers are $\mathbb{Z}^*$.
Set $\mathbb{Z}_+$: The non-negative integers are $\mathbb{Z}_+$.
Set $\mathbb{Z}_+^*$: The positive integers are $\mathbb{Z}_+^*$.

\[
\mathbb{Z} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \}
\]

Note that in this notation a subscripted + indicates positiveness but does not exclude a zero; and a superscripted * indicates exclusion of zero explicitly.

Theorem 7.1 (Some properties of integers). Let $a, b, c \in \mathbb{Z}$. The following properties are true.

- $a, b \in \mathbb{Z} \Rightarrow a \leq b$ or $b \leq a$ (total order)
- $a \leq a$ (reflexive property)
- $a \leq b$ and $b \leq a \iff a = b$ (antisymmetric property)
- $a \leq b$ and $b \leq c \Rightarrow a \leq c$ (transitive property)
- $a \geq b \Rightarrow a + c \geq b + c$ (transitive property)
- $a \geq b$ and $c \geq 0 \Rightarrow ac \geq bc$ (addition and multiplication closed over $\mathbb{Z}_+$)
- $a \geq 0$ and $b \geq 0 \Rightarrow ab \geq 0$ and $a + b \geq 0$ (addition and multiplication closed over $\mathbb{Z}_+$)

Theorem 7.2 (Properties of integers). Let $a, b, c \in \mathbb{Z}$. The following properties are true.

(The last or is disjunctive, not exclusive.)

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

An implication of the last property is that for $a, b, c \in \mathbb{Z}$, if $ab = ac$ and $a \neq 0$, then $a(b - c) = 0$ and since $a \neq 0$ then $b - c = 0$ i.e. $b = c$.

Definition 7.1 (Factorial $n!$ definition). 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.

Definition 7.2 (Choose symbol). The choose symbol as in $n$ choose $k$ or choose $k$ objects out of $n$.

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!}
\]
8 Collections: Sets

A collection of elements is a grouping of its elements. Two collections are of note: sets and sequences.

**Definition 8.1 (Set).** A set is a collection of elements in no particular order. The elements of a set are listed one after the other in no particular order separated by commas; curly braces { and } are used to denote and define the set. Thus \{10, 30, 20\} is a set of three elements.

**Definition 8.2 (Set ordering, and the belongs-to \(\in\) symbol).** In a set the order of its elements does not matter. Moreover a set does not contain duplicate elements thus \{10, 10\} is not a set. We say an element \(x\) belongs to \(A\) if and only if \(A\) contains \(x\); we write \(x \in A\).

**Definition 8.3 (Empty Set).** A set with no elements is empty. Thus \(\{}\) is an empty set. We also denote an empty set as \(\emptyset\) or \(\varnothing\). Thus \(\emptyset = \varnothing = \{}\).

**Definition 8.4 (Set enumeration and set comprehension; ellipsis).** The elements of a set can be enumerated individually and completely as in \{10, 30, 20\}, or through a comprehension that describes a property \(P(x)\) that is satisfied by them. Then the set of elements \(x\) satisfied by property \(P\) i.e. \(P(x)\) is described as \{\(x : P(x)\)\} or \{\(x \mid P(x)\)\}.

Infinite sets can use an ellipsis (three periods) as in \{1, 2, 3,\ldots\} or \{\ldots, 1, 2, 3,\ldots\} or \{\ldots, -2, -1, 0, 1, 2,\ldots\} for example.

**Definition 8.5 (Cardinality of a set).** The cardinality of a set is the number of its elements. The cardinality of set \(A\) is denoted as \(|A|\) or \(cA\) or \(c(A)\).

The empty set has cardinality zero. The cardinality of \{10, 30, 20\} or \(A = \{10, 30, 20\}\) is three. We write \(|A| = 3\) or \(c(A) = 3\) or \(cA = 3\).

**Definition 8.6 (Set equality).** Two sets \(A\) and \(B\) are equal to each other and we write \(A = B\) or \(B = A\) if and only if they have the same elements. 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 8.7 (Proper subsets and subsets).** A set \(A\) is called proper subset of \(B\) and denoted by \(A \subset B\) if every element of \(A\) is in \(B\) and \(B\) has at least one element not in \(A\). If the two sets are allowed to be equal we say \(A\) is a subset of \(B\) and write \(A \subseteq B\). Moreover \(A \subseteq B\) and \(B \subseteq A\) is equivalent to \(A = B\). Not a subset is denoted by \(\not\subset\) and \(\not\subseteq\) as needed.

**Definition 8.8 (Intersection and disjoint sets).** The intersection \(C\) of \(A\) and \(B\) is denoted by \(C = A \cap B\) and denotes the set that contains the common elements of \(A\) and \(B\). Thus \(C = A \cap B = \{x : x \in A \land x \in B\}\). Two sets \(E, F\) are disjoint if they have no common elements thus \(E \cap F = \emptyset\).

**Definition 8.9 (Union of sets).** The union \(D\) of \(A\) and \(B\) is denoted by \(D = A \cup B\) and denotes the set that contains all the elements of \(A\) and \(B\), but the common ones are listed once to comply with the definition of a set. Thus \(D = A \cup B = \{x : x \in A \lor x \in B\}\).
Definition 8.10 (PowerSet). The powerset of A signified as \( \mathcal{P}(A) \) or \( \mathcal{P}(A) \) is the set of all possible subsets of set A. Thus \( |\mathcal{P}(A)| = 2^{|A|} \).

For set A as previously defined \( \mathcal{P}(A) = \{\emptyset, \{10\}, \{20\}, \{10, 20\}, \{10, 30\}, \{20, 30\}, \{10, 20, 30\}\} \).

Definition 8.11 (Complement of a set). The complement of B sometimes denoted \( \overline{B} \) or \( B' \) or \( B^\complement \) is the set of elements not in B. For this to make sense, there should be a reference set call it A and all sets such as B are subsets of A. Then we can define the complement of B relative to A. Formally \( B^\complement = \{x \in A : x \notin B \} \).

Definition 8.12 (Relative complement and Absolute complement). For two sets A, B we define the relative complement of B relative to A to be the difference \( A - B \) or \( A \setminus B \) that is defined as \( A - B = A \setminus B = \{ x : x \in A \land x \notin B \} \). Thus \( A - B = A \cap B^\complement \). The absolute complement of B relative to a reference set that contains all sets including B is \( B^\complement = F - B \), where F is the reference set. Sometimes the latter is defined equivalently as \( A^\complement = (A - B) \cup (B - A) \).

Definition 8.13 (De Morgan’s Law). For every two sets that are subsets of a reference set F, we have \( (A \cup B)^\complement = A^\complement \cap B^\complement \) and \( (A \cap B)^\complement = A^\complement \cup B^\complement \).

Definition 8.14 (Complement Law and Involution). Moreover by De Morgan’s Law we can also derive that \( A \cup A^\complement = F \), \( A \cap A^\complement = \emptyset \) and \( \emptyset^\complement = F \) and \( F^\complement = \emptyset \). Moreover \( A \subseteq B \) implies \( B^\complement \subseteq A^\complement \). Finally \( A^\complement^\complement = A \).

Definition 8.15 (Ordered Pair, Triples and n-tuple). For a pair of elements a, b order matters. Thus \( (a, b) \) means that a is the first element of the pair and b is the second. Moreover \( (a, b) \neq (b, a) \). Likewise \( (a, b, c) \) and \( (x_1, \ldots, x_n) \) are triples/triplets and n-tuples respectively.

Definition 8.16 (Cartesian product; cartesian plane). For two sets A and B the cartesian product \( C = A \times B \) is defined as \( C = A \times B = \{(a, b) : a \in A, b \in B\} \). A cartesian product is generalizable to n sets, not just two. A cartesian plane is one where A = B = \( \mathbb{R} \).

Definition 8.17 (Natural numbers and Integer Numbers and Real Numbers). The set of integer numbers \( \mathbb{Z} \) includes positive and negative integers and zero. The set of Natural numbers or ordinal numbers includes the positive numbers and (but sometimes not) zero, i.e. it is the set of the non-negative integers or the set of positive integers. Thus to accommodate zero in the definition of natural numbers, we use \( \mathbb{N} \) and \( \mathbb{N}^+ \) to denote \( \mathbb{N}^+ = \mathbb{N} \setminus \{0\} \). Likewise \( \mathbb{R} \) is the set of real numbers.
9 Collections: Sequences

We can generalize the definition of a pair, triple or triplet, and \( n \)-tuple by defining a sequence in which elements can be repeated.

In a sequence, as opposed to a set, order matters, i.e. the elements of a sequence have some order.

**Definition 9.1 (Sequence).** A sequence is a collection of elements in which order matters i.e. it matters which element is first, second, etc. The elements of a sequence are listed according to their order and separated by commas; angular brackets \( ⟨ \) and \( ⟩ \) are used to denote and define a sequence.

**Exercise 9.1.** The first elements of sequence \( ⟨ 10, 30, 20 ⟩ \) is a 10, the second a 30, and the third a 20. This sequence is different from sequence \( ⟨ 10, 20, 30 ⟩ \). The two are different because for example the second element of the former is a 30, and the second element of the latter sequence is a 20. Thus those two sequences differ in their second element position. (They also differ in their third element position anyway.) Moreover we write \( ⟨ 10, 30, 20 ⟩ \not= ⟨ 10, 20, 30 ⟩ \) to indicate that they are not the same.

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 \( ⟨ 10, 10, 20 ⟩ \) is OK.

Sequences with too many elements to write down: ellipsis to the rescue. We can use the three period symbol known as an ellipsis (plural form, ellipses) that was introduced in the context of sets.

**Definition 9.2 (Sequence enumeration).** An infinite or finite sequence can also be described by a sequence comprehension or enumeration. For example \( ⟨a_i⟩ \) describes sequence \( a_1, a_2, \ldots, a_n \) or summarily \( a_i \) for \( i = 1, \ldots, n \), if we drop the angular brackets.

We can define sequence \( a_i = i \). This is sequence \( 1, 2, \ldots, n \) more correctly writtens as \( ⟨1, 2, \ldots, n⟩ \).

**Definition 9.3 (Sum of sequence and product of sequences).** The elements of a sequence can be summed or multiplied. Thus

\[
\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} i
\]

or inline \( \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} i \) and similarly \( \prod_{i=1}^{n} a_i = \prod_{i=1}^{n} i \) show how to denote this for the sequence of the previous example.
10 Real numbers, their properties and some of their functions

Set $\mathbb{R}$: The set of real numbers (or just reals) is denoted by $\mathbb{R}$.

All properties of integers translate to the domain of Real Numbers.

Moreover $\mathbb{R}$ with $\geq$ is a total order in the sense that

- (i) it is antisymmetric i.e. if $a \leq b$ and $b \leq a$ then $a = b$,
- (ii) it is transitive i.e. if $a \leq b$ and $b \leq c$ then $a \leq c$, and
- (iii) $a \leq b$ or $b \leq a$, and

moreover, (a) If $x \geq y$ then $x + z \geq y + z$ and (b) If $x \geq 0$ and $y \geq 0$ then $xy \geq 0$.

**Definition 10.1** (The floor function: $\lfloor x \rfloor$). The floor function is defined as follows: $\lfloor x \rfloor$ is the largest integer less than or equal to $x$.

**Definition 10.2** (The ceiling function: $\lceil x \rceil$). The ceiling function is defined as follows: $\lceil x \rceil$ is the smallest integer greater than or equal to $x$.

**Exercise 10.1** (The floor of $\lfloor 10.1 \rfloor$ and $\lfloor -10.1 \rfloor$). We have that $\lfloor 10 \rfloor$ is 10 itself. Moreover $\lfloor 10.1 \rfloor$ is 10 as well. Note that $\lfloor -10.1 \rfloor$ is $-11$.

**Exercise 10.2** (The ceiling of $\lceil 10.1 \rceil$ and $\lceil -10.1 \rceil$). We have that $\lceil 10 \rceil$ is 10 itself. Moreover $\lceil 10.1 \rceil$ is 11 as well. Note that $\lceil -10.1 \rceil$ is $-10$. 

11 Inequalities

For a discrete function the indeterminate (variable, unknown) will more often than not be denoted by \( n \). For continuous functions by \( x \). Thus \( A(n) \) is a discrete function and \( B(x) \) is a continuous variable function (unless otherwise noted). There are several ways one can use to compare two functions \( f(n) \) and \( g(n) \).

First we note the properties of integers as defined in Theorem 7.1 and Theorem 7.2. Those Theorems can be generalized in the domain of real numbers as well with some of them repeated in the previous page. With this in mind we enumerate some important ones here.

**Fact 11.1** (Inequalities).

\[
\begin{align*}
\text{Transitivity} & : a < b, \quad b < c \Rightarrow a < c. \\
R1. & : a < b, \quad \forall q \Rightarrow a + q < b + q \\
R2. & : a < b, \quad c < d \Rightarrow a + c < b + d \\
R3. & : a < b, \quad c > 0 \Rightarrow a \cdot c < b \cdot c \\
R4. & : a < b, \quad c < 0 \Rightarrow a \cdot c > b \cdot c \\
R5. & : 0 < a < b \Rightarrow 1/a > 1/b \\
R6. & : \text{Flip} < \text{into} > \text{and} > \text{into} <
\end{align*}
\]

**Example 11.1.** Show that for \( 0 < a < b \) we have that \( a^2 < b^2 \).

*Proof.*

Use R3 with \( c = a \land a > 0 \):

\[
a < b \Rightarrow a^2 < ab \quad (1)
\]

Use R3 with \( c = b \land b > 0 \):

\[
a < b \Rightarrow ab < b^2 \quad (2)
\]

Use Transitivity and (1) and (2) above:

\[
a^2 < ab \land ab < b^2 \Rightarrow a^2 < b^2
\]

**Example 11.2.** Show that for \( 0 < a < b \) we have that \( a^3 < b^3 \).

**Example 11.3.** Show that \( \sum_{i=1}^{n} i^3 \leq n^4 \).

*Proof.*

\[
\sum_{i=1}^{n} i^3 = 1^3 + 2^3 + 3^3 + \ldots + i^3 + \ldots + (n-1)^3 + n^3.
\]

Since \( i \) runs from 1 to \( n \) we have, using the examples before,

\[
\begin{align*}
1 \leq i \leq n : 1 & \leq n \Rightarrow 1^3 \leq n^3 \\
\ldots & \ldots \\
1 \leq n & \Rightarrow i^3 \leq n^3 \\
\ldots & \ldots \\
n \leq n & \Rightarrow n^3 \leq n^3
\end{align*}
\]

Add both sides of inequalities

\[
1^3 + 2^3 + \ldots + i^3 + \ldots + n^3 \leq n^3 + n^3 + \ldots + n^3 + \ldots + n^3
\]

\[
\sum_{i=1}^{n} i^3 \leq n \cdot n^3 = n^4
\]

\[\square\]
12 The factorial and Stirling’s approximation formula

Theorem 12.1 (Upper bound for factorial). For every \( n \geq 1 \), we have that \( n! \leq n^n \).

Proof. The first result is a straightforward upper bound of each term of the factorial with \( n \).

\[
    n! = \underbrace{1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n}_{n \text{ terms}} \\
    \leq \underbrace{n \cdot n \cdot \ldots \cdot n \cdot n}_{n \text{ terms}} \\
    \leq n^n
\]

Corollary 12.1 (Logarithmic Upper bound). If \( n! \leq n^n \) by taking logarithms \( \lg(n!) \leq n \lg n \).

Theorem 12.2 (Lower bound for factorial). For every \( n \geq 1 \), we have \( n! > (n/2)^{(n/2)} \).

Proof. The smallest half of the terms of the factorial are at least one (lower bound). The largest half of the terms of the factorial are at least \( n/2 \). Moreover \( \lceil n/2 \rceil + \lfloor n/2 \rfloor = n \) and \( \lfloor n \rfloor \leq n \leq \lceil n \rceil \). Therefore,

\[
    n! = \underbrace{1 \cdot 2 \cdot \ldots \cdot \lceil n/2 \rceil \cdot \lfloor n/2 \rfloor \cdot \ldots \cdot (n-1) \cdot n}_{n \text{ terms}} \\
    \geq \underbrace{1 \cdot 1 \cdot \ldots \cdot \lfloor n/2 \rfloor \cdot \ldots \cdot \lceil n/2 \rceil \cdot \lfloor n/2 \rceil}_{\lceil n/2 \rceil \text{ terms}} \\
    \cdot \underbrace{\lfloor n/2 \rfloor \cdot \ldots \cdot \lceil n/2 \rceil}_{\lfloor n/2 \rfloor \text{ terms}} \\
    \geq (\lfloor n/2 \rfloor)^{\lceil n/2 \rceil} \\
    \geq (n/2)^{(n/2)}.
\]

Corollary 12.2 (Logarithmic Lower bound). If \( n! > (n/2)^{(n/2)} \) by taking logarithms \( \lg(n!) \geq (n/2) \lg(n/2) \).

Example 12.1. For the floor or ceiling manipulations consider \( n = 6 \) with \( n/2 = 3 \) and \( \lceil n/3 \rceil = \lfloor n/3 \rceil = 2 \).

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

Likewise, for \( n = 5 \) with \( n/2 = 2.5 \) and \( \lceil 5/2 \rceil = 2, \lfloor 5/2 \rfloor = 2, \).

\[
    5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \geq 1 \cdot 1 \cdot 1 \cdot 3 \cdot 3 \cdot 3 = 3^3 \geq (5/2)^{(5/2)}
\]

Theorem 12.3 (Stirling’s approximation formula for \( n! \)). For all \( n \geq 10 \), we have that

\[
    n! = \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \left( 1 + \Theta \left( \frac{1}{n} \right) \right).
\]
Corollary 12.3 (Simplified Stirling’s formula). For \( n \geq 10 \) we have that \( n! \approx \left( \frac{n}{e} \right)^n \).

Corollary 12.4. A corollary is that \( \lg(n!) = \Theta(n \lg n) \) after the \( \Theta \) is formally introduced.

Proof. From either Stirling’s theorem or its Corollary, we have that

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

\( \square \)
13 Limits, Derivatives and Integrals

Fact 13.1 (Limits in computing). In computing limits are used for asymptotic considerations and are for \( n \rightarrow \infty \) i.e. for asymptotically large values of some integer parameter \( n \). The parameter \( n \) is sometimes referred to as problem size and less often as input size. Derivatives are thus \( \frac{d}{dn} \).

Fact 13.2 (Simple derivatives). For any constant \( k > 0 \), we have that

\[
(k)' = 0, \quad (n^k)' = k \cdot n^{k-1}, \quad (e^n)' = e^n, \quad (a^n)' = a^n \cdot \ln(a), \quad (\ln(n))' = \frac{1}{n},
\]

\[
(2^n)' = 2^n \cdot \ln(2) \approx 2^n, \quad (\lg(n))' \approx \frac{1}{n}
\]

Fact 13.3 (Multiplication and Division).

\[
(f(n)g(n))' = f'(n)g(n) + f(n)g'(n), \quad \left( \frac{f(n)}{g(n)} \right)' = \frac{f'(n)g(n) - f(n)g'(n)}{g^2(n)}
\]

Fact 13.4 (Integrals).

\[
\int n^k \, dn = n^{k+1}/(k+1) + c, \quad \int (1/n) \, dn = \ln(|n|) + c, \quad \int (e^n) \, dn = e^n + c, \quad \int (a^n) \, dn = a^n/\ln(a) + c,
\]

Fact 13.5 (Limits). For any constant \( k > 0 \),

\[
\lim_{n \to \infty} n = \infty, \quad \lim_{n \to \infty} n \ln n = \infty, \quad \lim_{n \to \infty} n^k = \infty, \quad \lim_{n \to \infty} \ln n = \infty, \quad \lim_{n \to \infty} 2^n = \infty
\]

Fact 13.6.

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

Fact 13.7.

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

Example 13.1 (L’Hospital). For infinite limits, if L’Hospital is to be applied, constants from base conversions are less of an issue and can be ignored without affecting the end result. Thus in the third equality do not spend time thinking of whether \( 1/n \) must be multiplied with \( \ln 2 \) or divided by \( \ln 2 \) or something else.

\[
\lim_{n \to \infty} \frac{n \ln n}{n^2} = \lim_{n \to \infty} \frac{\ln n}{n} = \lim_{n \to \infty} \frac{(\ln n)'}{(n)'} = \lim_{n \to \infty} \frac{1/n}{1} = 0
\]
14 Minimum and Maximum of a function: \( f(n) \)

**Fact 14.1** (Extremes). Finding minima or maxima of a function involves taking the first derivative, equating it to zero and solving for the indeterminate to determine the critical point(s) of the function. Then finding the sign of the second derivative at the critical points.

**Example 14.1** (Maximum). What is the maximum of \( f(t) = t^2 + (n - 1 - t)^2 \) in the interval \([0, n - 1]\)?

**Proof.** Find the first derivative \( f'(t) \), equate it to zero and solve for the indeterminate \( t \) to determine the critical point. \( f'(t) = 2t + 2(n - 1 - t)(-1) \Rightarrow f'(t) = 0 \Rightarrow t = (n - 1)/2. \)

The second derivative \( f''(t) = 2 + 2(-1)(-1) = 4 > 0 \). The function has a minimum at \( t = (n - 1)/2 \). We are not interested in its minimum but rather in its maximum. It is a parabola and thus it is symmetric in the range \([0, n - 1]\). The maximum would be at the extreme \( t = 0 \) or \( t = n - 1 \) or there might be two maxima at \( t = 0, t = n - 1 \). Indeed

\[ f(0) = f(n - 1) = 0^2 + (n - 1 - 0)^2 = (n - 1)^2 + (n - 1 - (n - 1))^2 = (n - 1)^2. \]

\[ \square \]

**Example 14.2** (Minimum). What is the minimum of \( f(t) = t \log(t) + (n - 1 - t) \log(n - 1 - t) \) in the interval \([0, n - 1]\)?

**Proof.** Find the first derivative \( f'(t) \).

\[ f'(t) = 1 \cdot \log(t) + t \cdot \left(1/t\right) + (-1) \log(n - 1 - t) + (n - 1 - t)(-1/(n - 1 - t)) = \log(t) - \log(n - 1 - t) \Rightarrow f'(t) = \log(t/(n - 1 - t)). \]

Solving for \( t \) in \( f'(t) = 0 \) relates to \( \log(t) - \log(n - 1 - t) = 0 \) i.e. \( t = n - 1 - t \Rightarrow t = (n - 1)/2 \). Finding the second derivative of \( f(t) \) i.e. the derivative of \( \log(t) - \log(n - 1 - t) \) gives

\[ f''(t) = (\log(t) - \log(n - 1 - t))' = 1/t - (-1)/(n - 1 - t) = 1/t + 1/(n - 1 - t) > 0 \]

Thus the \( t = (n - 1)/2 \) is a minimum.

As a conclusion

\[ f\left(\frac{n-1}{2}\right) = (n - 1) \log((n - 1)/2) = (n - 1) \log(n - 1) - (n - 1). \]

\[ \square \]
15 Sums of sequences: finite and infinite

Fact 15.1 (Some constants). The base $e$ of the Neperian (also known as natural) logarithms, $\pi$, Euler’s gamma constant, Golden ratio $\phi$ and its Fibonacci-related counter-part are shown.

\[ e \approx 2.718281, \quad \pi \approx 3.14159, \quad \gamma \approx 0.57721, \quad \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61803, \quad \hat{\phi} = \frac{1 - \sqrt{5}}{2} \approx -0.61803. \]

Definition 15.1 (Sums and Sigma notation). The sum $a_0 + a_1 + \ldots + a_n$ can be represented in compact form as

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

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 $a_i = i^2, a_i = i^3$.

Theorem 15.1 (Binomial Theorem). For all $n \in \mathbb{N}$ and $a, b$ we have

\[ (a + b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k = a^n + na^{n-1}b + \frac{n(n-1)}{2}a^{n-2}b^2 + \ldots + \binom{n}{k} a^{n-k}b^k + \ldots + nab^{n-1} + b^n. \]

A series is the sum of a sequence.

Fact 15.2 (Arithmetic series). The arithmetic series is the sum of the (terms of the) arithmetic sequence i.e. the sum of the consecutive integers starting from one (or zero) to an upper bound usually denoted by $n$. A closed-form expression is shown.

\[ A(n) = A_n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \]

Fact 15.3 (Quadratic and Cubic series). They are defined analogously. A closed-form expression is shown.

\[ Q(n) = Q_n = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \quad C(n) = C_n = \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}. \]

Fact 15.4 (Harmonic series).

\[ H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \int \frac{1}{x} dx \approx \ln(n) + \gamma. \]
Fact 15.5 (Derivative formulae from harmonic series). Let \( H_n = \sum_{i=1}^{n} 1/i \). Then we have the following equations.

\[
\sum_{i=1}^{n} H_i = (n+1)H_n - n, \quad \sum_{i=1}^{n} iH_i = \frac{n(n+1)}{2}H_n - \frac{n(n-1)}{4}.
\]

The following identities can be derived from the following theorem that is proved by induction, as special cases for \( n = 2 \) and \( n = 3 \), and with the third identity deriving from the second by substituting \(-b\) for \( b\).

Fact 15.6. For all \( a, b \)

\[
a^2 - b^2 = (a - b)(a + b), \quad a^3 - b^3 = (a - b)(a^2 + ab + b^2), \quad a^3 + b^3 = (a + b)(a^2 - ab + b^2).\]

Theorem 15.2. For all \( n \in \mathbb{N} \) and \( a, b \) we have

\[
(a^n - b^n) = (a - b) \left( a^{n-1} + a^{n-2}b + a^{n-3}b^2 + \ldots + ab^{n-2} + b \right).
\]

Theorem 15.3 (Geometric series). For all \( n \in \mathbb{N} \) and \( a \) we have

\[
(a^n - 1) = (a - 1) \left( a^{n-1} + a^{n-2} + a^{n-3} + \ldots + a + 1 \right).
\]

This is the previous theorem for \( b = 1 \) and can be rewritten in the form of the geometric series provided that \( a \neq 1 \).

\[
G(n, a) = G^n = \sum_{i=0}^{n-1} a^i = 1 + a + a^2 + \ldots + a^{n-1} = \frac{a^n - 1}{a - 1}.
\]

Theorem 15.4 (Infinite geometric series). If \( |a| < 1 \) to the limit \( n \to \infty \) we have the following

\[
\lim_{n \to \infty} 1 + a + a^2 + \ldots + a^{n-1} = \lim_{n \to \infty} \frac{a^n - 1}{a - 1} = \frac{1}{1-a}.
\]

Therefore

\[
G(a) = G^a = \sum_{i=0}^{\infty} a^i = \frac{1}{1-a}.
\]

Fact 15.7. For \( a \neq 1 \) we have the following finite sum.

\[
I(n, a) = I^n = \sum_{i=0}^{n-1} ia^i = \frac{(n-1)a^{n+1} - na^n + a}{(1-a)^2}.
\]

Correspondingly, the infinite sum is derived as follows.

Fact 15.8. For \( |a| < 1 \) we have the following sum.

\[
I(a) = I^a = \sum_{i=1}^{\infty} ia^i = \frac{a}{(1-a)^2}.
\]
Corollary 15.1 \((I(n, 1/2))\). Moreover, we have that for \(I(n, 1/2)\),

\[
I(n, 1/2) = \sum_{i=0}^{n-1} i/2^i = 2 - \frac{2n + 2}{2^n}.
\]

\[
I(1/2) = \sum_{i=0}^{\infty} i/2^i = 2.
\]

Fact 15.9. For \(|a| < 1\).

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

\[
\ln(1 + a) = a - \frac{a^2}{2} + \ldots + \frac{(-1)^{i+1}a^i}{i} + \ldots = \sum_{i=1}^{\infty} \frac{(-1)^{i+1}a^i}{i}.
\]

\[
\ln \frac{1}{1-a} = a + \frac{a^2}{2} + \ldots + \frac{a^i}{i} + \ldots = \sum_{i=1}^{\infty} \frac{a^i}{i}.
\]

Fact 15.10. For \(|a| < 1\), and \(F_i\) the Fibonacci sequence,

\[
\frac{1}{(1-a)^{n+1}} = \sum_{i=0}^{\infty} \binom{i+n}{i} a^i,
\]

\[
\frac{1}{\sqrt{1-4a}} = \sum_{i=0}^{\infty} \binom{2i}{i} a^i,
\]

\[
\frac{a}{1-a-a^2} = a + a^2 + 2a^3 + 3a^4 + \ldots = \sum_{i=0}^{\infty} F_i a^i.
\]

Fact 15.11.

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad \sum_{k=0}^{n} \binom{n}{k} = 2^n, \quad \binom{n}{k} = \binom{n}{n-k}, \quad \binom{n}{k} = \binom{n}{k} + \binom{n-1}{k-1}.
\]
Proof. (Theorem 15.2) Since $A(n) = A_n = 1 + 2 + \ldots + (n - 1) + n$. Writing $A_n$ forwards and backwards we add up the corresponding terms.

\[
A_n = 1 + 2 + \ldots + (n - 1) + n \\
A_n = n + (n - 1) + \ldots + 2 + 1 \quad \text{Add up this and the previous equation}
\]

\[
2A_n = (n + 1) + (n + 1) + \ldots + (n + 1) + (n + 1)
\]

\[
A_n = n(n + 1)/2
\]

Proof. (Theorem 15.2, Theorem 15.3) We can use the following method to find sums of the following form

\[
S_k = \sum_{i=1}^{n} i^k.
\]

First consider $(i + 1)^{k+1}$ and expand it. Substitute in the expansion $i = 1, i = 2, \ldots, i = n$, a total of $n$ times and write the resulting $n$ equalities one after the other. Then, sum these $n$ equalities by summing up the left hand sides and the right hand sides. Solve for $S_k$ and $S_k$ can then be found as a function of $n$.

For the sum in question $k = 1$. Therefore we consider

\[
(i + 1)^2 = i^2 + 2i + 1
\]

We substitute for $i = 1, 2, \ldots, n$ writing one equality after the other

\[
(1 + 1)^2 = 1^2 + 2 \cdot 1 + 1 \\
(2 + 1)^2 = 2^2 + 2 \cdot 2 + 1 \\
(3 + 1)^2 = 3^2 + 2 \cdot 3 + 1 \\
(4 + 1)^2 = 4^2 + 2 \cdot 4 + 1 \\
\ldots = \ldots \\
(n + 1)^2 = n^2 + 2 \cdot n + 1
\]

When we sum up the $n$ equalities we realize that say, $(3 + 1)^2$ of the third line is equal to $4^2$ of the fourth line and therefore.

\[
(1 + 1)^2 + (2 + 1)^2 + \ldots + (n + 1)^2 = (1^2 + 2^2 + 3^2 + \ldots + n^2) + 2 \cdot (1 + 2 + \ldots + n) + (1 + \ldots + 1)
\]

We note that $2 \cdot (1 + 2 + \ldots + n) = 2S_1$ and $(1 + \ldots + 1) = n$ (number of ones is number of equations). Then,

\[
(n + 1)^2 = 1 + 2S_1 + n
\]

Solving for $S_1$ we get that $S_1 = ((n + 1)^2 - n - 1)/2$, ie $S_1 = (n^2 + 2n + 1 - n - 1)/2 = (n^2 + n)/2 = n(n + 1)/2$, which is $A(n) = A_n$.  \qed
Proof. (Fact 15.8, Corollary 15.1) We show that

\[ I(1/2) = I^{1/2} = \sum_{i=0}^{\infty} i/2^i = 2. \]

We start with the geometric series and in particular the infinite geometric series. Getting its first derivative with respect to \( a \) and then multiplying both sides with \( a \) yields, almost, the result. In the last step we substitute \( 1/2 \) for \( a \). In all cases \( |a| < 1 \).

\[
G(n, a) = \sum_{i=0}^{n-1} a^i = 1 + a + \ldots + a^i + \ldots + a^{n-1} = \frac{a^n - 1}{a - 1}
\]

\[
G(a) = \sum_{i=0}^{\infty} a^i = 1 + a + \ldots + a^i + \ldots = \frac{1}{1-a}
\]

\[
I(a) = \sum_{i=0}^{\infty} i \cdot a^i = 1 \cdot a^1 + 2 \cdot a^2 + \ldots + i \cdot a^i + \ldots = ?
\]

For any \( |a| < 1 \), we have.

\[
\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}
\]

\[
1 + a + \ldots + a^i + \ldots = \frac{1}{1-a}
\]

\[
(1 + a + \ldots + a^i + \ldots)' = \left( \frac{1}{1-a} \right)'
\]

\[
0 + 1 \cdot a^0 + \ldots + i \cdot a^{i-1} + \ldots = \frac{1}{(1-a)^2}
\]

\[
0 \cdot a + 1 \cdot a^1 + \ldots + i \cdot a^i + \ldots = \frac{a}{(1-a)^2}
\]

\[
I(a) = \sum_{i=0}^{\infty} i \cdot a^i = \frac{a}{(1-a)^2}
\]

From the last one for \( a = 1/2 \) we get \( I(1/2) = 2 \). \( \square \)
16  Comparison of (discrete or continuous) functions

For a discrete function the indeterminate (variable, unknown) will more often than not be denoted by \( n \). For continuous functions by \( x \). Thus \( A(n) \) is a discrete function and \( B(x) \) is a continuous variable function (unless otherwise noted). There are several ways one can use to compare two functions \( f(n) \) and \( g(n) \).

**Method 16.1** (Using direct or indirect methods and transitivity: from \( F(n) \) to \( G(n) \)). One can compare two functions \( F(n) \) and \( G(n) \), by using inequalities and starting with one (say, \( F(n) \)) the other (i.e. \( G(n) \)) is derived (see previous page). Or indirectly as follows.

1. Show first that \( F(n) < F_1(n) \) where \( F_1(n) \) is an 'easier to deal' function.
2. Then show that \( G_1(n) < G(n) \) likewise.
3. Then show also directly or using latter methods or indirectly (recursively) that \( F_1(n) < G_1(n) \).
4. Two applications of transitivity show that \( F(n) < F_1(n) \) and \( F_1(n) < G_1(n) \) imply \( F(n) < G(n) \).

**Method 16.2** (Difference \( F(n) – G(n) \)). One can compare two functions \( F(n) \) and \( G(n) \), by taking their difference \( F(n) – G(n) \) or \( G(n) – F(n) \) and determining the sign of the difference.

**Method 16.3** (Ratio \( F(n)/G(n) \)). One can compare two functions \( F(n) \) and \( G(n) \), by taking their difference \( F(n)/G(n) \) or \( G(n)/F(n) \) and determining whether it is greater, equal or less than one.

**Method 16.4** (Raise using common base and then compare i.e. \( 2^{F(n)} : 2^{G(n)} \)). One can compare two functions \( F(n) \) and \( G(n) \), by comparing \( 2^{F(n)} \) and \( 2^{G(n)} \).

**Method 16.5** (Take logarithms to same base and then compare i.e. \( \lg F(n) : \lg G(n) \)). One can compare two functions \( F(n) \) and \( G(n) \), by comparing \( \lg F(n) \) to \( \lg G(n) \).

**Example 16.1** (Transitivity). The two functions \( f(n) = 25n^2 \) and \( g(n) = 25n^2 + 10 \) can be compared by starting say from \( f(n) \)

\[
f(n) = 25n^2 < 25n^2 + 1 < 25n^2 + 2 < \ldots < 25n^2 + 10 = g(n)
\]

(Transitivity is implicitly used \( A < B < C \) means \( A < B \) and \( B < C \) and thus by transitivity \( A < C \).)

**Example 16.2** (Difference). The two functions \( f(n) = 25n^2 \) and \( g(n) = 25n^2 + 10 \) can be compared by taking the difference \( g(n) – f(n) = 25n^2 + 10 – 25n^2 = 10 > 0 \) and observing that \( g(n) – f(n) \) is positive i.e. \( g(n) > f(n) \). Equivalently \( f(n) – g(n) < 0 \) and also \( f(n) < g(n) \).

**Example 16.3** (Ratio). The two functions \( f(n) = 25n^2 \) and \( g(n) = 25n^2 + 10 \) can be compared by taking the ratio \( g(n)/f(n) = (25n^2 + 10)/25n^2 \geq 1 \) and observing that \( g(n)/f(n) > 1 \) i.e. \( g(n) > f(n) \). Equivalently \( f(n)/g(n) < 1 \) and also \( f(n) < g(n) \).

**Example 16.4** (Exponentiation). Let \( a(n) = \ln(25n^2) \) and \( b(n) = \ln(25n^2 + 10) \). Then \( f(n) = e^{a(n)} = 25n^2 \) and \( g(n) = e^{b(n)} = 25n^2 + 10 \) The resulting functions \( f(n) \) and \( g(n) \) have already been compared and found to be such that \( f(n) < g(n) \). Thus \( a(n) < b(n) \) as well by monotonicity!
Example 16.5 (Logarithms). The two functions \( f(n) = 25n^2 \) and \( g(n) = 25n^2 + 10 \) can be compared by taking their logarithms \( a(n) = \ln(25n^2) \) and \( b(n) = \ln(25n^2 + 10) \). We just proved above that \( a(n) < b(n) \) and thus \( f(n) < g(n) \) by monotonicity. (Admittedly, not a very interesting example!)

Example 16.6. Compare \( n^{1/\lg n} \) and 2.

Proof. See also Exercise 2.1 for solution or

\[
\begin{align*}
n^{1/\lg n} & : 2 \\
\lg (n^{1/\lg n}) & : \lg 2 \\
(1/\lg n) \cdot \lg n & : 1 \\
1 & : 1.
\end{align*}
\]

Obviously 1 = 1 and this \( n^{1/\lg n} \) is equal to 2. \( \square \)
17 Asymptotics

**Fact 17.1** (Polynomial functions). A function \( n^m \) for any positive constant integer \( m > 0 \) is a polynomial in \( n \) i.e. a polynomial function. The linear combination of polynomial functions is also a polynomial function.

**Fact 17.2** (Polylogarithmic functions). A function \( \lg^m n \) for any positive constant integer \( m > 0 \) means \( (\lg n)^m \) and is a polylogarithmic function.

**Fact 17.3** (Large enough means asymptotically large). The expression “for large enough \( n \)” means “there is a positive constant \( n_0 \) such that for all \( n > n_0 \),” that is for “asymptotically large values of \( n \”).

**Fact 17.4** (Polynomial vs Polylogarithmic). For any positive constant integer \( k, m > 0 \) and integer \( n > 0 \), we have that \( n^m > \lg^k n \) for large enough \( n \). This means that every polynomial function is asymptotically larger than any logarithmic function of the same variable \( n \).

**Fact 17.5** (Exponential vs Polynomial). For any positive constant integer \( m \) and integer \( n > 0 \), we have that \( 2^n > n^m \) for large enough \( n \). This means that every exponential function (base two) is asymptotically larger than any polynomial function of the same variable \( n \).

The results above are true even for non-integer but positive constant values for \( k, m \). Any such constant value is bounded between two integer constant values.

**Fact 17.6** (Linear, Log-linear, Quadratic, Cubic). A linear functions is \( n \), a quadratic function is \( n^2 \) and cubic \( n^3 \). A log-linear functions is \( n\lg n \) the product of a linear and a logarithmic function. A linear function is asymptotically smaller than a log-linear function which is likewise smaller than a quadratic function and likewise (asymptotically) smaller than a cubic function.

A log-linear function can be considered a polynomial function as \( n < n\lg n < n^2 \) for large enough \( n \). Note also that \( n\lg n = n^{1+\lg\lg n/\lg n} \).
18 Asymptotic Comparison: informal (and dangerous) approach

Definition 18.1 (Asymptotic comparison of functions: an informal overview). An asymptotic comparison of two functions is not an exact comparison. We do not care to establish what happens between \( f(n) \) and \( g(n) \) for all values of \( n \). We care only how the two functions compare asymptotically i.e. for large \( n \). This means that asymptotic comparisons is of interest for \( n \to \infty \).

Definition 18.2 (Asymptotic comparison of functions: an informal definition). For two functions \( f(n) \) and \( g(n) \) that are always positive, the \( \lim_{n \to \infty} f(n)/g(n) \) establishes whether \( f(n) \) is asymptotically larger, equal or smaller than \( g(n) \) depending on whether the limit is \( \infty \), a constant other than zero, and zero respectively.

Definition 18.3 (Informal (no limits) asymptotic comparison). A "naive way" to compare asymptotically two functions \( f(n), g(n) \) involves eliminating low-order terms and multiplicative constant terms. The leftovers say \( F(n) \) and \( G(n) \) are then directly compared using \( < \), \( > \), \( = \), \( \leq \), \( \geq \). A direct comparison of the \( F(n) \) and \( G(n) \) does not imply a same-like comparison for \( f(n) \) and \( g(n) \): only an asymptotic comparison is then possible. Therefore special symbols are introduced as asymptotic equivalents to \( < \), \( > \), \( = \), \( \leq \), \( \geq \). These are \( o \), \( \omega \), \( \Theta \), \( O \), \( \Omega \) respectively.

\[
\begin{align*}
S1. \text{Eliminate low order terms first} & \quad f(n) = 256n^2 + 20n + 3 : g(n) = 128n^2 - 3 \\
S2. \text{Multiplicative constants become 1} & \quad 256n^2 + 20n \quad : \quad 128n^2 \\
S3. \text{Compare directly the leftovers} & \quad F(n) = 1 \cdot n^2 \quad = \quad G(n) = 1 \cdot n^2 \\
S4. \text{asymptotic symbol in column 1} & \quad f(n) \quad = \quad \Theta(G(n)) \\
S4. \text{(alternative formulation)} & \quad f(n) \quad \in \quad \Theta(G(n))
\end{align*}
\]

AsymptoticComparison  DirectComparison
\[
\begin{align*}
f(n) : g(n) & \quad F(n) : G(n) \\
\Theta & = \\
O & \leq \\
\Omega & \geq \\
o & < \\
\omega & >
\end{align*}
\]
19 Asymptotic Comparison: formal definitions

Two functions \( f(n) \) and \( g(n) \) are formally asymptotically compared as follows

**Definition 19.1 (Asymptotic comparison of \( f(n) \) and \( g(n) \) using \( o, \omega, \Theta \)).** In order to asymptotically compare two always positive functions \( f(n) \) and \( g(n) \) with the three symbols \( o, \omega, \Theta \), we first find the limit \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \) and then determine which case is applicable as follows. (\( c \) is a positive constant).

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \begin{cases} 
0 & \Rightarrow f(n) = o(g(n)) \\
\infty & \Rightarrow f(n) = \omega(g(n)) \\
c \neq 0 & \Rightarrow f(n) = \Theta(g(n))
\end{cases}
\]

**Corollary 19.1 (\( o \Rightarrow O \) implication).** If \( f(n) = o(g(n)) \), then \( f(n) = O(g(n)) \).

**Corollary 19.2 (\( \omega \Rightarrow \Omega \) implication).** If \( f(n) = \omega(g(n)) \), then \( f(n) = \Omega(g(n)) \).

**Corollary 19.3 (\( \Theta \Leftrightarrow \Omega \cap O \)).** \( f(n) = \Theta(g(n)) \), if and only if \( f(n) = \Omega(g(n)) \) and \( f(n) = O(g(n)) \).

**Fact 19.1 (Asymptotically smaller, equal, large).** When \( f(n) = o(g(n)) \) we say that \( f(n) \) is asymptotically smaller than \( g(n) \).

When \( f(n) = \omega(g(n)) \) we say that \( f(n) \) is asymptotically larger than \( g(n) \).

When \( f(n) \equiv \Theta(g(n)) \) we say that \( f(n) \) is asymptotically equal to \( g(n) \).

Naturally in the former case \( g(n) \) is asymptotically larger than \( f(n) \), in the second case above \( g(n) \) is asymptotically smaller than \( f(n) \). In the third case \( g(n) \) is also asymptotically equal to \( f(n) \).

**Fact 19.2 (Comparison vs Asymptotic comparison).** When we compare two functions \( f(n) \) and \( g(n) \) we directly compare them using \(<, > \) etc. When we asymptotically compare two functions \( f(n) \) and \( g(n) \) we take the limit of \( \lim_{n \to \infty} f(n)/g(n) \) and then we use the asymptotic symbols \( o, \omega, \Theta \) depending on whether the limit is \( 0, \infty \) or some non-zero (and positive) constant.

**Example 19.1 (Asymptotically equal does not mean equal).** 5 and 6 as number are not the same, 5 is smaller than 6 i.e. 5 is not equal to 6. But \( f(n) = 5 \) and \( g(n) = 6 \) means that ‘5’ and ‘6’ are both constant functions. They are asymptotically equal as \( \lim_{n \to \infty} 5/6 \) is a non-zero constant. Thus functions 5 and 6 are asymptotically equal (and by functions we mean \( f(n) = 5 \) and \( g(n) = 6 \)).

**Example 19.2.** Functions \( 5n \) and \( 6n \) are asymptotically equal. Their limit is \( 5/6 \) (or \( 6/5 \) the other way around). They are both linear functions.

**Example 19.3.** Functions \( 5n^2 \) and \( 6n^2 \) are asymptotically equal. Their limit is \( 5/6 \) (or \( 6/5 \) the other way around). They are both quadratic functions.

**Example 19.4.** Function \( n \) is asymptotically larger than \( 1,000,000,000,000 \). Their limit is \( \infty \). (It does not matter what happens if \( n = 1 \) or \( n = 1000 \) or \( n = 1,000,000 \). It only matters what happens for \( n \to \infty \).)
20 Asymptotic notation

Definition 20.1 (Little-o). \( f(n) = o(g(n)), \text{ iff } \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0. \)

Definition 20.2 (Little-omega). \( f(n) = \omega(g(n)), \text{ iff } \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty. \)

Definition 20.3 (Big-Oh). \( f(n) = O(g(n)) \text{ iff } \exists \text{ positive constants } c_2, n_0 : 0 \leq f(n) \leq c_2 g(n) \forall n \geq n_0. \)

Corollary 20.1 (Big-Oh: If little-o then Big-Oh.). If \( f(n) = o(g(n)), \) then \( f(n) = O(g(n)). \)

Definition 20.4 (Big-Omega). \( f(n) = \Omega(g(n)) \text{ iff } \exists \text{ positive constants } c_1, n_0 : 0 \leq c_1 g(n) \leq f(n) \forall n \geq n_0. \)

Corollary 20.2 (Big-Omega: If little-omega then Big-Omega.). If \( f(n) = \omega(g(n)), \) then \( f(n) = \Omega(g(n)). \)

Definition 20.5 (Theta). \( f(n) = \Theta(g(n)) \text{ iff } \exists \text{ positive constants } c_1, c_2, n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0. \)

Definition 20.6 (Theta –Limit definition). \( f(n) = \Theta(g(n)), \text{ iff } \lim_{n \to \infty} \frac{f(n)}{g(n)} = c, \text{ where } c > 0 \text{ is a (positive) constant (and other than zero).} \)

Corollary 20.3 (Theta: Big-Oh and Big-Omega if-and-onlyif Theta.). \( f(n) = \Theta(g(n)), \text{ iff } f(n) = \Omega(g(n)) \) and \( f(n) = O(g(n)). \)
**Note 20.1** (Set theoretic view of $O,a,Θ,ω,Ω$). When we say $n^2 + 10 = O(n^2)$ we view $O(n^2)$ as the set containing all quadratic functions. Thus it also contains $n^2 + 10$ which is indeed a quadratic function. Likewise $5 = O(1)$ means 5 belongs to the set of all constant functions. Some textbooks are using the notation $n^2 + 10 \in O(n^2)$ or $5 \in O(1)$ instead of using the equal sign. And as a side note, $5 = Θ(1)$ or $5 \in Θ(1)$ is a better tighter description and so is $n^2 + 10 = Θ(n^2)$ or $n^2 + 10 \in Θ(n^2)$.

$O(n^2)$ includes not only quadratic functions, but also linear, log-linear, logarithmic and everything else in-between or asymptotically smaller.

**Note 20.2** (Non-commutative use of symbols). $Θ(1) = 5$ means nothing. The symbols as defined earlier appear always to the right of the equal sign. The set theoretic view make it easy to argue that $Θ(1) \in 5$ does not make sense.

**Note 20.3** (Tomatoes vs Potatoes or < and $O$ etc). $1 \leq O(n)$ is meaningless. One can say $1 = O(n)$ and this was properly defined above, or $1 = o(n)$. Nothing else was defined involving the $<$, $>$, $\leq$, $\geq$ and the five letter symbols!

**Fact 20.1** (Polynomial vs Polylogarithmic: an asymptotic comparison). Some obvious results (constant $m,k > 0$): $n^m = ω(\lg^k n)$. This derives from the fact that in general $n^m \geq \lg^k n$ for any positive constant $m,k$ and large enough $n$.

**Fact 20.2** (Exponential vs polynomial: an asymptotic comparison). Some obvious results (constant $m > 0$): $2^n = ω(n^m)$. This derives from the fact that in general $2^n > n^m$ for any positive constant $m$ and large enough $n$.

**Fact 20.3** (Factorial). Some obvious results (constant $m > 0$): $n! = ω(n^m)$. This derives from the fact that $n! > n^m$ for any constant $m$ and large enough $n$.

**Fact 20.4** (Log-factorial). Some obvious results (constant $k$): $\lg (n!) = Θ(n \lg n)$. Also, a result of Stirling’s approximation formula for the factorial.

**Example 20.1.** Which of $a_0 + a_1 n + a_2 n^2 + a_3 n^3$ and $n^2$ is asymptotically larger, where $a_i > 0$ for all $i$?

**Proof.** Consider $a_0 + a_1 n + a_2 n^2 + a_3 n^3$. As all $a_i$ are positive, then $a_0 > 0$ and $a_1 n > 0$ and $a_2 n^2 > 0$ and thus $a_0 + a_1 n + a_2 n^2 + a_3 n^3 > a_3 n^3$.

We now show that our lower bound $a_3 n^3$ is $Ω(n^2)$. As $a_3 > 0$, it is obvious that $a_3 n^3 > 1 \cdot n^2$ for any $n > 1/a_3$ (note that $a_3 > 0$ DOES NOT MEAN THAT $a_3 > 1$, as $a_3$ is real and not necessarily an integer).

Therefore for $c = 1$ and $n_0 = 1/a_3$ we have shown that $a_0 + a_1 n + a_2 n^2 + a_3 n^3 = Ω(n^2)$.

**Example 20.2.** Which of the two functions is asymptotically larger $a_0 + a_1 n + a_2 n^2 + a_3 n^3$ or $n^4$, where $a_i > 0$ for all $i$?

**Proof.** **Hint.** When we intend to prove $f(n) = O(g(n))$, as is the case here, it sometimes helps to find an upper bound $h(n)$ for $f(n)$ i.e such that $f(n) \leq h(n)$ and then show that $h(n) = O(g(n))$. Since in $a_0 + a_1 n + a_2 n^2 + a_3 n^3$ all $a_i$ are positive we take the maximum of all $a_i$ and we call it $A$. Then we have that $a_i < A$ for all $i$. Also, $An^i < An^3$ for $i \leq 3$. Then

$$a_0 + a_1 n + a_2 n^2 + a_3 n^3 \leq A + An + An^2 + An^3 \leq An^3 + An^3 + An^3 + An^3 = 4An^3$$
Finally $4A n^3 \leq n^4$ for all $n > 4A$.

We have shown that

$$a_0 + a_1 n + a_2 n^2 + a_3 n^3 \leq 4A n^3 \leq 1 \cdot n^4$$

for all $n \geq n_0 = 4A$, where $A$ is the maximum of $a_0, a_1, a_2, a_3$. As all $a_i$ are constant, so is $4A$. Therefore the constants in the $O$ definition are $c = 1$ and $n_0 = 4A$, where $A = \max\{a_0, a_1, a_2, a_3\}$. 

Exercise 20.1. Do the Exercises of the textbook for the chapters/sections covered in class. The more you do of them the more you practice.

Exercise 20.2. Show that

$$\sum_{i=1}^{n} i^2 = \Theta(n^3).$$

What are the values of $c_1, c_2$ and $n_0$? Justify your answer.

Exercise 20.3. TRUE or FALSE?

1. $\lg(n!) = O(n^2)$.
2. $n + \sqrt{n} = O(n^2)$.
3. $n^2 + \sqrt{n} = O(n^2)$.
4. $n^3 + 2\sqrt{n} = O(n^2)$.
5. $1/n^3 = O(\lg n)$.
6. $n^2 \sin^2(n) = \Theta(n^2)$. ($\sin$ is the well-known trigonometric function).

Exercise 20.4. Prove the following.

1. $(n - 10)^2 = \Theta(n^2)$.
2. $n^4 + 10n^3 + 100n^2 + 1890n + 98000 = \Omega(n^4)$.
3. $n^4 + 10n^3 + 100n^2 + 1890n + 98000 = \Omega(n^2)$.
4. $n^4 - 10n^3 - 100n^2 - 1890n + 100000 = O(n^4)$.
5. $n^2 - 20n - 20 = \Omega(n)$.
6. $n^2 + 20n = O(n^2)$. 
21 Recurrences

21.1 Master Method

The master method is one of three methods to solve recurrence relations i.e. recursive formulae/expressions. The solution however it provides is an asymptotic one and it works only for recurrences of a certain form.

Method 21.1 (Master Method). Let \( T(n) = aT(n/b) + f(n) \) be 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 \( f(n) = O(n^{\log_b a - \varepsilon}) \) for some constant \( \varepsilon > 0 \), then \( T(n) = \Theta(n^{\log_b a}) \).
- **M2** If \( f(n) = \Theta(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a} \log n) \).
- **M3** If \( f(n) = \Omega(n^{\log_b a + \varepsilon}) \) for some constant \( \varepsilon > 0 \), and if \( a f(n/b) \leq c f(n) \) for some constant \( 0 < c < 1 \) and for large \( n \), then \( T(n) = \Theta(f(n)) \).

Method 21.2 (Master Method alternative formulation of case M2). There is an alternative formulation for Case 2 (aka M2) of the master method.

- **M2'** If \( f(n) = \Theta(n^{\log_b a} (\log n)^k) \), for some non-negative constant \( k \), then \( T(n) = \Theta(n^{\log_b a} (\log n)^{k+1}) \).
21.2 Substitution Method

The second method for solving a recurrence is known as the substitution method. Here we know what the solution would look like either through trial of error or some prior knowledge. We would like to verify indeed that this is a solution. Sometimes the method is known as guess and check or guess and verify. The checking or verify part involves the use of Induction and in particular Strong Induction. The guess part can be an exact one, an asymptotic one (Big-Oh, or little-oh, Big-Omega, or little-omega) or a tight one (Theta).

11.2 Substitution Method: MergeSort recurrence (Upper Bound).

Exercise 21.1 (Merge-Sort recurrence). Solve the recurrence \( T(n) = 2T(n/2) + n \) using the substitution (a.k.a. guess-and-check) method. (Implicit assumption is that \( T(n) \) is nonnegative and defined for all positive \( n \), or for arbitrarily large \( n \)).

Hint: Show that \( T(n) = O(n \log n) \).

Proof. By (strong) induction after we guess the answer. Since no boundary condition is given we can thus choose \( k \) and \( l \) constants greater than zero so that \( T(k) = l \). We choose \( k, l \) in such a way to make the inductive proof as simple as possible. Let us choose \( T(1) = 0 \).

A. Guess Step: \( T(n) = O(n \log n) \). We guess \( T(n) = O(n \log n) \), i.e. \( \exists \) pos. constant \( c_2, n_0 : T(n) \leq c_2 n \log n \), \( \forall n \geq n_0 \).

A.0 Check Step (Strong Induction). We shall prove our claim by using induction. Our \( P(n) \), the proposition to be proven true would be \( T(n) \leq c_2 n \log n \) for arbitrarily large values of \( n \geq n_0 \), for some positive constant \( n_0 \).

A.1. Base Case of Induction. We show \( P(1) \) is true is \( T(1) \leq c_2 \cdot 1 \cdot \log 1 \). Since \( T(1) = 0 \) it is trivially true that \( 0 = T(1) \leq c_2 \cdot 1 \cdot \log 1 = 0 \) for all choices of a positive and constant \( c_2 \). Base case completed.

A.2. Inductive Step. \( P(1) \land \ldots \land P(n/2) \land \ldots \land P(n - 1) \Rightarrow P(n) \). We show that if \( P(i) \) is true for all \( i < n \) then we will show that \( P(n) \) is also true. \( P(n) \) is \( T(n) \leq c_2 \cdot n \cdot \log n \). If \( P(i) \) is true for all \( i < n \), since \( n/2 < n \) then \( P(n/2) \) is also true. Then by the inductive assumption for \( i = n/2 \) we have \( T(n/2) \leq c_2 (n/2) \log (n/2) \).

We then use the recurrence and utilize the inductive hypothesis for \( T(n/2) \)

\[
T(n) = 2T(n/2) + n \\
= 2 \underbrace{T(n/2)}_{P(n/2): T(n/2) \leq c_2(n/2) \log (n/2)} + n \\
\leq 2c_2(n/2) \log (n/2) + n = c_2 n \log n - c_2 n + n
\]

To complete the inductive step we must show that the rightmost expression \( c_2 n \log n - c_2 n + n \) is at most \( c_2 n \log n \). For that to be true i.e. \( c_2 n \log n - c_2 n + n \leq c_2 n \log n \) we need \( -c_2 n + n < 0 \) i.e. \( c_2 \geq 1 \).

Thus if we choose a \( c_2 = 1 \) which is \( \geq 1 \) and for the \( n_0 \) of the base case i.e. \( n_0 = 1 \) we have that

\[
T(n) \leq n \log n \quad \forall n \geq 1.
\]
Exercise 21.2 (Lower bound). For the previous example can you also determine a lower bound?

**Proof.** **B. Guess Step:** \( T(n) = \Omega(n \lg n) \). We guess \( T(n) = \Omega(n \lg n) \), i.e. \( \exists \text{ pos. constant } c_1, n_0 : T(n) \geq c_1 n \lg n, \forall n \geq n_0 \).

**B.0 Check Step (Strong Induction).** We shall prove our claim by using induction. The \( P(n) \) is \( T(n) \geq c_1 n \lg n \) for arbitrarily large values of \( n \geq n_0 \), for some positive constant \( n_0 \).

**B.1. Base Case of Induction.** We show \( P(1) \) is true: \( T(1) \geq c_1 \cdot 1 \cdot \lg 1 \). Since \( T(1) = 0 \), it is true that \( 0 = T(1) \geq c_1 \cdot 1 \cdot \lg 1 = 0 \) for all choices of a positive and constant \( c_1 \). Base case completed.

**B.2. Inductive Step.** \( P(1) \land \ldots \land P(n/2) \land \ldots \land P(n-1) \Rightarrow P(n) \). We show that if \( P(i) \) is true for all \( i < n \) then we will show that \( P(n) \) is also true. \( P(n) \) is \( T(n) \geq c_1 \cdot n \cdot \lg n \). If \( P(i) \) is true for all \( i < n \), since \( n/2 < n \) then \( P(n/2) \) is also true. Then by the inductive assumption for \( i = n/2 \) we have \( T(n/2) \geq c_1 (n/2) \lg (n/2) \).

We then use the recurrence and utilize the inductive hypothesis for \( T(n/2) \).

\[
T(n) = 2T(n/2) + n
\]

\[
P(n/2): T(n/2) \geq c_1 (n/2) \lg (n/2)
\]

\[
= 2 \cdot \frac{T(n/2)}{\lg 1} + n
\]

\[
\geq 2c_1 (n/2) \lg (n/2) + n = c_1 n \lg n - c_1 n + n
\]

To complete the inductive step we must show that the rightmost expression \( c_1 n \lg n - c_1 n + n \) is at least \( c_1 n \lg n \). For that to be true i.e \( c_1 n \lg n - c_1 n + n \geq c_1 n \lg n \) we need \(-c_1 n + n > 0 \) i.e. \( c_1 \leq 1 \).

Thus if we choose a \( c_1 = 1 \) which is \( \leq 1 \) and for the \( n_0 \) of the base case i.e. \( n_0 = 1 \) we have that

\[
T(n) \geq n \lg n \quad \forall n \geq 1.
\]

Well we have just proven that \( T(n) \geq n \lg n \). We also proved (previous page) that \( T(n) \leq n \lg n \). Both of them prove that \( T(n) = n \lg n \) ! (The last mark is an exclamation mark, not a factorial!)

Thus while our objective was to prove a tight asymptotic bound the derivation of the three constants \( n_0, c_1, c_2 \) allowed us to determine an exact bound.

\[\square\]

Exercise 21.3 (Same example: Direct method). Solve the recurrence by guessing an exact answer.

**Proof.** **C. Guess Step:** \( T(n) = An \lg n + Bn + C \). Let us guess an answer that contains a log-linear, a linear and a constant term. Let’s try to compute the constant values \( A, B \) and \( C \). Only condition is that \( A \) should be positive.

**C.1. Base Case utilization.** We have \( T(1) = 0 \) substituting 1 for \( n \) in \( T(n) = An \lg n + Bn + C \) we get \( 0 = T(1) = A \cdot 1 \cdot \lg 1 + B \cdot 1 + C \) i.e. \( B + C = 0 \).

**C.2. Recurrence utilization.** Since \( T(n) = 2T(n/2) + n \) we have that

\[
T(n) = 2T(n/2) + n
\]

\[
An \lg n + Bn + C = 2(An/2 \lg (n/2) + Bn/2 + C) + n
\]

\[
An \lg n + Bn + C = An \lg n + (B - A + 1)n + 2C
\]
In the last equality the first term of both sides cancels out. We then equate the coefficients of \( n \) and the constant terms and solve for \( A, B, C \) also utilizing the base case \( B + C = 0 \).

The constant terms \( C \) and \( 2C \) should be equal to each other for \( C = 0 \). Since \( B + C = 0 \) we also have \( B = 0 \).

The linear term coefficients \( B \) and \( (B - A + 1) \) should be equal to each other \( B = B - A + 1 \). This results to an \( A = 1 \). Thus the guessed function \( T(n) = An \lg n + Bn + C \) gets resolved to having \( A = 1 \) and \( B = C = 0 \) leading to the same result with the iteration or previous substitution method. No need to explicitly do induction.
21.3 Iteration or Recursion tree method

If the method is done graphically it is sometimes called the recursion tree method (see textbook, or the class proof of the running time of merge-sort). The traditional name is iteration method.

How. Expand the recurrence all the way down to the base case and sum up the residuals. In order to avoid problems with floors and ceilings and the use of identities such as that for all integers \(n, a, b\), we have \(\lfloor \frac{n}{ab} \rfloor = \lfloor \frac{n}{a} \rfloor / \lfloor \frac{n}{b} \rfloor\), we are going to make an assumption that \(n\) is a power of 2 and thus \(n/2, n/4, n/8\) etc are all integer.

Exercise 21.4. Solve \(T(n) = 2T(n/2) + n\), \(T(1) = 0\).

Proof. In order to compute \(T(n)\) we need to establish \(T(n/2)\). Likewise in order to compute \(T(n/2)\) we need to establish \(T(n/4)\). We iterate \(i\) iterations until \(T(n/2^i)\) with the intent of making \(n/2^i = 1\). If \(n/2^i = 1\) then we know that \(T(1) = 0\). For \(n/2^i = 1\) we have \(n = 2^i\) and solving for \(i\) we get \(i = \log_2 n\).

\[
T(n) = 2T(n/2) + n = 2\left(2T(n/2^2) + n/2\right) + n
\]

\[
= 2^2T(n/2^2) + 2n = 2^2\left(2T(n/2^3) + n/2^2\right) + 2n
\]

\[
= 2^3T(n/2^3) + 3n
\]

\[
\ldots
\]

\[
= 2^iT(n/2^i) + i \cdot n \quad \text{Now, Substitute} \quad i = \log_2 n
\]

\[
= 2^{\log_2 n} \cdot T(n/2^{\log_2 n}) + \log_2 n \cdot n = n \cdot T(1) + \log_2 n \cdot n = n \log_2 n
\]

- Keep track of number of iterations/depth of recursion tree.
- Keep track of sum of terms per iteration/level of recursion tree.
- Sometimes, the two previous steps and experience allow us to guess the solution correctly. We can then stop the solution with this method and switch to the substitution method instead.

Example 21.1. Solve exactly \(T(n) = 2T(n/2) + n\), \(T(2) = 1\). Assume \(n\) is a power of 2.

The previous example was straightforward and uncomplicated. Let’s go back to the merge-sort recurrence we solved in class through the recursion tree method. (The one whose last term is \(n - 1\) not \(n\) referring to the maximum number of comparisons performed to merge two sorted sequences each of size \(n/2\).)
Exercise 21.5. Solve $T(n) = 2T(n/2) + n - 1$, \quad $T(1) = 0$.

Proof. Again we shall terminate recursion for $T(n/2^i)$ when $n/2^i = 1$ and solving for $i$ we shall get $i = \lg n$. We shall also need the geometric sequence result $2^0 + 2^1 + \ldots + 2^{i-1} = 2^i - 1$.

\[
T(n) = 2T(n/2) + n - 1 \\
= 2(2T(n/2^2) + n/2 - 1) + n - 1 = 2^2T(n/2^2) + 2n - 1 - 2 \\
= 2^2(2T(n/2^3) + n/2^2 - 1) + 2n - 1 - 2 = 2^3T(n/2^3) + 3n - 2^0 - 2^1 - 2^2 \\
\vdots \\
= 2^iT(n/2^i) + i \cdot n - 2^0 - 2^1 - \ldots - 2^{i-1} = 2^iT(n/2^i) + i \cdot n - (2^i - 1) \\
= 2^iT(n/2^i) + i \cdot n - (2^i - 1) \\
\text{Now, Substitute} \quad i = \lg n \\
= 2^{\lg n} \cdot T(n/2^{\lg n}) + \lg n \cdot n - (2^{\lg n} - 1) \\
= n \cdot T(1) + \lg n \cdot n - n + 1 = n\lg n - (n - 1)
\]

The previous page recurrence was generating an additive term which was the same in each iteration: $n$. This one generates a term that is iteration dependent: $n - 2^{i-1}$ for iteration $i$. Thus for $i = 1$ we get $n - 1$, for $i = 2$ we get $n - 2$, and so on.

The answer for $T(n)$ with the $-(n - 1)$ low-order term is the one we got through the recursion tree method as well. \hfill \Box

Exercise 21.6. Solve the recurrence $T(n) = 8T(n/2) + n$ using the iteration/recursion tree method. Assume that $T(1) = 5$.

Proof.

\[
T(n) = 8T(n/2) + n \\
= 8(8T(n/2^2) + n/2) + n \\
= 8^2T(n/2^2) + 8n/2 + n \\
= 8^2T(n/2^2) + (8/2)^1n + (8/2)^0n \\
= 8^2(8T(n/2^3) + n/2^2) + (8/2)^1n + (8/2)^0n \\
= 8^3T(n/2^3) + 8^2n/2^2 + (8/2)^1n + (8/2)^0n \\
= 8^3T(n/2^3) + (8/2)^2n + (8/2)^1n + (8/2)^0n \\
= \ldots \\
= 8^iT(n/2^i) + (8/2)^{i-1}n + \ldots + (8/2)^1n + (8/2)^0n
\]

Again the boundary case is $T(1) = 5$. We set $n/2^i = 1$, ie $i = \lg n$. Then for $i = \lg n$, $T(n/2^i) = T(1) = 5$. 
We therefore get for $i = \lg n$.

$$
T(n) = 8T(n/2) + n
= 8^i T(n/2^i) + (8/2)^{i-1} n + \ldots + (8/2)^1 n + (8/2)^0 n
= 8^{\lg n} T(n/2^{\lg n}) + \frac{(8/2)^{\lg n} - 1}{8/2 - 1} \cdot n
= 2^{3\lg n} T(1) + \frac{(4)^{\lg n} - 1}{3} \cdot n
= n^3 T(1) + \frac{(4)^{\lg n} - 1}{3} \cdot n
= 5n^3 + \frac{n^2 - 1}{3} n
$$

To verify our calculations we observe that $T(1) = 5 + (1 - 1)/3 = 5$ and

$$
T(n) = 8T(n/2) + n
= 8(5(n/2)^3 + \frac{(n/2)^2 - 1}{3} n) + n
= 5n^3 + \frac{n^2 - 1}{3} n
= T(n),
$$

ie the recurrence is verified. □
22 Frequency and the Time domain

Definition 22.1 (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 1ms, 1µs, 1ns, 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 22.2 (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. \(1\text{Hz} = 1\text{cycle/s}\).

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 22.3 (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 \(f=5\text{Hz}\) 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 modern CPU rated at 2GHz allows instructions to be completed in \(1/2,000,000,000 = 0.5\text{ns}\).

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

Definition 22.4 (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 ‘‘inanosecond’’.
23 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 23.1 (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 23.1 (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 23.1 (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 23.2 (Denary natural numbers in fixed-width). An n-digit radix-10 natural integer number a is denoted as \(a = a_{n-1}a_{n-2}...a_0\), where \(a_i \in \{0,\ldots,9\}\) for all \(0 \leq i < n\). The most-significant digit is \(a_{n-1}\) and the least-significant digit is \(a_0\). The magnitude of the number is

\[
|a| = \sum_{i=0}^{i=n-1} a_i \cdot 10^i.
\]

The value of \(a\) is its magnitude i.e. \(a = |a|\).

The definition can easily extend to integer numbers in general.

Definition 23.3 (Denary integer numbers in fixed-width). An n-digit radix-10 natural integer number a is denoted as \(a = sa_{n-1}a_{n-2}...a_0\), where \(a_i \in \{0,\ldots,9\}\) for all \(0 \leq i < n\), and s is + or empty to indicate a positive integer, empty for zero, or − to indicate a negative integer. The most-significant digit is \(a_{n-1}\) and the least-significant digit is \(a_0\), and s is the sign. The magnitude of the number is

\[
|a| = \sum_{i=0}^{i=n-1} a_i \cdot 10^i.
\]

The value of \(a\) is \(a = (-1) \cdot |a|\) if the sign of \(a\) is \(a = -1\), or its magnitude \(a = |a|\) otherwise.
Example 23.2 (Units of radix-10 integer). For \( a = a_n a_{n-1} \ldots a_0 \), \( a_i \in \{0, \ldots, 9\} \) for all \( 0 \leq i < n \), the digit \( a_i \) 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 \( a_0 \) is the number of units, \( a_1 \) is the number of tens, \( a_2 \) is the number of hundreds, \( a_3 \) is the number of thousands contributing to the magnitude of \( a \).

Method 23.1 (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.

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

Example 23.3 (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 \( 1230000 \).

<table>
<thead>
<tr>
<th>Base or Radix</th>
<th># digits</th>
<th>digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>2</td>
<td>0, 1</td>
</tr>
<tr>
<td>Octal</td>
<td>8</td>
<td>0 .. 7</td>
</tr>
<tr>
<td>Denary</td>
<td>10</td>
<td>0 .. 9</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>16</td>
<td>0 .. 9 , a .. f ; alternative: 0 .. 9 , A .. F</td>
</tr>
</tbody>
</table>

Fact 23.1 (Table of integers). A table of some integers in binary, octal, hexadecimal and denary is shown below.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Denary</th>
<th>Hexadecimal</th>
<th>Octal</th>
<th>Shorthand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
<td>10</td>
<td>1000</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>1001</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>A</td>
<td>12</td>
<td>1010</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>B</td>
<td>13</td>
<td>1011</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>C</td>
<td>14</td>
<td>1100</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>D</td>
<td>15</td>
<td>1101</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>E</td>
<td>16</td>
<td>1110</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>F</td>
<td>17</td>
<td>1111</td>
</tr>
<tr>
<td>10000</td>
<td>16</td>
<td>10</td>
<td>20</td>
<td>10000</td>
</tr>
<tr>
<td>10001</td>
<td>17</td>
<td>11</td>
<td>21</td>
<td>10001</td>
</tr>
</tbody>
</table>
**Definition 23.4** (Binary notation of a natural number). A natural number \( a \) is denoted in radix-2 as the \( n \)-digit (or \( n \)-bit) sequence \( \text{bin}(a) = a_{n-1}a_{n-2}\ldots a_0 \), where \( a_i \in \{0, 1\} \) for all \( 0 \leq i < n \). For \( \text{bin}(a) \) in binary notation its magnitude and value \( a \) is

\[
a = |a| = \sum_{i=0}^{i=n-1} a_i \cdot 2^i.
\]

**Definition 23.5** (Octal notation of a natural number). A natural number \( a \) is denoted in radix-8 as the \( n \)-digit sequence \( \text{oct}(a) = a_{n-1}a_{n-2}\ldots a_0 \), where \( a_i \in \{0, 1, 2, 3, 4, 5, 6, 7\} \) for all \( 0 \leq i < n \). For \( \text{oct}(a) \) in octal notation its magnitude \( a \) is

\[
a = |a| = \sum_{i=0}^{i=n-1} a_i \cdot 8^i.
\]

**Definition 23.6** (Hexadecimal notation of a natural number). A natural number \( a \) is denoted in radix-16 as the \( n \)-character sequence \( \text{hex}(a) = a_{n-1}a_{n-2}\ldots a_0 \), where \( a_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\} \) for all \( 0 \leq i < n \), or equivalently \( a_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\} \) for all \( 0 \leq i < n \). For \( \text{hex}(a) \) in hexadecimal notation its magnitude \( a \) is

\[
a = |a| = \sum_{i=0}^{i=n-1} a_i \cdot 16^i,
\]

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 23.4.** If we write 101 we might indicate \( \text{101}_{10} \) or \( \text{101}_2 \) or \( \text{101}_8 \) or \( \text{101}_{16} \).

- \( \text{101}_{10} \) is \( 101 \) in denary which is one hundred and one.
- \( \text{101}_2 \) is \( 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5 \) i.e. five in denary.
- \( \text{101}_8 \) is \( 1 \cdot 8^2 + 0 \cdot 8^1 + 1 \cdot 8^0 = 65 \) i.e. sixty-five in denary.
- \( \text{101}_{16} \) 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 23.5 (Convert binary into denary).** Find the magnitude or value \( x \) of the 5-bit binary number \( (n = 5) \) with \( \text{bin}(x) = 11001 \).

\[
\begin{array}{cccccc}
2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
1 & 1 & 0 & 0 & 1 \\
\end{array}
\]

\[
16 + 8 + 0 + 0 + 1 = 25
\]

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 23.6 (Fixed-width vs Minimal-width).

\[
\begin{align*}
0, 1 & \text{ represented in binary with } 1 \text{ bit as } 0, 1 \\
0, 1, 2, 3 & \text{ represented in binary with } 2 \text{ bits as } 00, 01, 10, 11 \\
0, 1, 2, 3, 4, 5, 6, 7 & \text{ represented in binary with } 3 \text{ bits as } 000, 001, 010, 011, 100, 101, 110, 111 \\
0 - 15 & \text{ represented in binary with } 4 \text{ bits as } 0000 - 1111 \\
& \vdots \\
0 \cdots 2^m - 1 & \text{ represented in binary with } m \text{ bits as } 00 \cdots 0 \underbrace{\cdots 1 \cdots 1}_m \text{ bits } m \text{ bits }
\end{align*}
\]

Example 23.7 (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 23.8 (m ones). The value \(a\) of m-bit bin(a) = \(1 \cdots 1\) is \(a = 2^m - 1\).

Example 23.9 (One followed by \(m - 1\) zeroes). The value \(a\) of m-bit bin(a) = \(0 \underbrace{1 \cdots 1}_m\) is \(a = 2^{m-1}\).

Definition 23.7 (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 23.8 (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 23.10 (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 23.11.

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, because digits A,B are radix-16 digits only

Fact 23.2 (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 \(m = \lfloor \lg a \rfloor + 1 = \lfloor \lg (a + 1) \rfloor\).

Thus for 1 we need 1 bit, for 2 i.e. 10\(_2\) we need two, for 4 i.e. 100\(_2\) we need three and for 7 i.e 111\(_2\) we also need three.
Proof. From Example 23.8 and Example 23.9 we have that any natural number \( a \) such that \( 2^{m-1} \leq a \leq 2^m - 1 \) needs \( m \) bits for its representation \( \text{bin}(a) \). 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 23.6.

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

\[
\begin{align*}
2^{m-1} & \leq a \leq 2^m - 1, \\
2^{m-1} & \leq a < 2^m, \\
m - 1 & \leq \lfloor \log_a \rfloor \leq \log_a, \\
m - 1 & \leq \lfloor \log_a \rfloor \leq \log_a < m.
\end{align*}
\]

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

Similarly,

\[
\begin{align*}
2^{m-1} & \leq a \leq 2^m - 1, \\
2^{m-1} + 1 & \leq (a + 1), \\
2^{m-1} & \leq (a + 1) < 2^m, \\
m - 1 & \leq \log((a + 1)), \\
m - 1 & \leq \lfloor \log((a + 1)) \rfloor \leq \log((a + 1)) < m.
\end{align*}
\]

Since \( \lfloor \log((a + 1)) \rfloor > m - 1 \) and \( \lfloor \log((a + 1)) \rfloor \leq m \), there can be only one possibility that \( \lfloor \log((a + 1)) \rfloor = m \). \( \square \)

**Extension of Algorithm in Example 23.5 : Radix-b to Radix-10**

**Fact 23.3 (Radix-b to Radix-10).** 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**

**Algorithm Base-2-to-Base-10**

\[
\begin{align*}
\text{RES} = 0; \ b = 2; \\
\text{repeat until all bits are read} & : 0 \ast b + \, 1 \, \rightarrow \, 0 \quad > \, 1 \, 0 \, 1 \, 0 \, 1 \, 1 \\
\text{read_next_bit t; \# shown next to} & : 1 \ast b + \, 0 \, \rightarrow \, 2 \quad > \, 1 \, 0 \, 1 \, 0 \, 1 \, 1 \\
\text{RES = RES \ast b \, + \, t; \# RES next to} = & : 2 \ast b + \, 1 \, \rightarrow \, 5 \quad 1 \, 0 \, > \, 1 \, 0 \, 1 \, 1 \\
5 \ast b + \, 0 \, \rightarrow \, 10 & : 1 \, 0 \, 1 \, > \, 0 \, 1 \, 1 \\
10 \ast b + \, 1 \, \rightarrow \, 21 & : 1 \, 0 \, 1 \, 0 \, > \, 1 \, 1 \\
21 \ast b + \, 1 \, \rightarrow \, 43 & : 1 \, 0 \, 1 \, 0 \, 1 \, > \, 1
\end{align*}
\]
**Binary into Octal, Binary into Hexadecimal**

**Fact 23.4 (Radix-2 to Radix-8: Groups of 3 bit).** For natural number \(a\) for which \(\text{bin}(a)\) 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 23.1. (The leftmost group might have its binary digits padded with leading zeroes to have four bits.)

\[
\begin{align*}
1'1'1'1'1'1'1'1'1 & \quad \text{: Group into groups of 3 bits} \quad \text{: Step 1} \\
0'1'1'1'1'1'1 & \quad \text{: Add leading zeroes left group} \quad \text{: Step 2} \\
3'7'7 & \quad \text{: Convert triplets into octal} \quad \text{: Step 3} \\
00377 & \quad \text{: Output} \quad \text{: Step 4}
\end{align*}
\]

**Fact 23.5 (Radix-2 to Radix-16: Groups of 4 bit).** For natural number \(a\) for which \(\text{bin}(a)\) 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 23.1. (The leftmost group might have its binary digits padded with leading zeroes to have three bits.)

\[
\begin{align*}
1'1'1'1'1'1'1'1'1 & \quad \text{: Group into groups of 4 bits} \quad \text{: Step 1.} \\
1'1'1'1'1'1'1'1 & \quad \text{: Add leading zeroes left group} \quad \text{: Step 2} \\
F'F & \quad \text{: Convert quadruplets into hex} \quad \text{: Step 3} \quad \text{[Use also Table of Fact 6.1]} \\
0XFF & \quad \text{: Output using A-F} \quad \text{: Step 4} \\
or
0xff & \quad \text{: Output using a-f} \quad \text{: Step 4}
\end{align*}
\]

**Fact 23.6 (Radix-10 to Radix-2: Right to Left).**

**Input:** Decimal integer \(a\).

**Output:** Binary representation of \(\text{bin}(a)\) 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 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 \(X = 0\) go to Step 5, else go to Step 2 and repeat.

**Step 5.** Output the result (write it down properly).

**Fact 23.7 (Radix-10 to Radix-2: Left to Right).**

**Input:** Decimal integer \(a\).

**Output:** Binary representation of \(\text{bin}(a)\) 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 23.9 \((\text{bin}(x,n))\). We shall use the notation \(\text{bin}(x,n)\) to denote the \(n\)-bit notation of denary \(x\) in binary. (Leading zeroes are used to pad the result to \(n\) bit.)

Definition 23.10 \((\text{oct}(x,n))\). We shall use the notation \(\text{oct}(x,n)\) 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.)

Definition 23.11 \((\text{hex}(x,n)\) or \(\text{HEX}(x,n))\). We shall use the notation \(\text{hex}(x,n)\) or \(\text{HEX}(x,n)\) 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.)

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

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

**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 23.2 this is \(\log n\) if we substitute \(a=n-1\) in Fact 23.2, 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.)

<table>
<thead>
<tr>
<th>Divisor</th>
<th>Dividend</th>
<th>; Q= Divisor / Dividend</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>8</td>
<td>in-denary</td>
</tr>
<tr>
<td>2**8</td>
<td>2**3</td>
<td>in-denary but base 2 exp notation</td>
</tr>
<tr>
<td>Q= 2**8</td>
<td>/</td>
<td>2<strong>3 = 2</strong>(8-3)= 2**5 = 32 in denary</td>
</tr>
<tr>
<td>bin(256)=100000000</td>
<td>bin(8)= 1000</td>
<td>in-binary</td>
</tr>
<tr>
<td>Q= SHIFTRIGHT(100000000,3)=100000</td>
<td>Q is 100000 in binary (i.e. 2**5)</td>
<td></td>
</tr>
</tbody>
</table>

**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 \(0 \leq L < n\), \(0 \leq P < n/s\), and \(0 \leq T < s\).

There is an easy way to obtain \(P\) and \(T\) from \(L\):
\[
P = \lfloor L/s \rfloor, \quad T = L \mod s.
\]
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 23.12** *(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:
\[
(n, s) : \quad L = (P, T) \Rightarrow P = \lfloor L/s \rfloor, \quad T = L \mod s, \quad 0 \leq L < n, \quad 0 \leq P < n/s, \quad 0 \leq T < s.
\]
In C/C++ \(\lfloor \cdot \rfloor\) is integer division and \(\mod\) is denoted \%. Thus another way to write it is to say
\[
(n, s) : \quad L = (P, T) \Rightarrow P = (L/s), \quad T = L\%s, \quad 0 \leq L < n, \quad 0 \leq P < n/s, \quad 0 \leq T < s.
\]

**Definition 23.13** *(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 \(L\).
\[
(n, s) : \quad (P, T) = L \Rightarrow L = P \times s + T, \quad 0 \leq L < n, \quad 0 \leq P < n/s, \quad 0 \leq T < s.
\]

**Example 23.12.** *(To make initial calculations easy, we drop the power of two requirement.)* If we have a memory of size \(n = 100,000\) bytes and a paged organization with page size \(s = 5,000\) bytes, then we can view memory as a collection of 20 pages \((n/s = 100000/5000 = 20)\) each of size \(s = 5000\) bytes. Thus an \(L = 23456\) gets mapped to \(P = 23456/5000 = 4\), 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 23.14.** We view a logical address $L$ as the concatenation of a page number $P$ and an offset $T$. Thus $L = (P,T)$ becomes $L = \langle P,T \rangle$, where $\langle \rangle$ is the concatenation operator.

<table>
<thead>
<tr>
<th>$P$: Page Number</th>
<th>$T$: Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$: Logical Address</td>
<td></td>
</tr>
</tbody>
</table>

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 23.15.** 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.

<table>
<thead>
<tr>
<th>$P$: 5-bit Page Number</th>
<th>$T$: 3-bit Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$: 8-bit Logical Address</td>
<td></td>
</tr>
</tbody>
</table>

**Example 23.13.** 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 $0 \leq P < G = 2^5 = 32$.

\[
\begin{array}{cccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & \end{array}
\]  

$\{ \text{Logical Address} \}$

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-most 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 $L = P \times s + T = 11 \times 8 + 1 = 89$. 
24  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 24.1 and also in Example 24.2 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 24.1 and the Unicode representation of the same letter A.

Example 24.1.

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

Example 24.2.

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

Example 24.3.

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

Example 24.4.

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

Fact 24.1 (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 24.5 (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 24.1. The ordinal value of that byte, viewed as an unsigned integer, is 65.

Example 24.6 (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 24.3. 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 24.4. Thus symbol 1 has a different ordinal value than the magnitude of the binary representation of natural number one.
Fact 24.2 (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 24.3 (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 24.2 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 24.4 (Java char). In Java the char data type has size 2B; Java uses UTF-16 representation. It can only represent 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 24.5 (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-significant 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 24.7 (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 24.6 (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 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 ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
3 @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
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 DEL
```

NUL = null  BS = Backspace  DLE = Datalink escape  CAN = cancel
SOH = start of heading  TAB = horizontal tab  DC1 = Device control 1  EN = end of medium
STX = start of text  LF = linefeed/newline  DC2 = SUB = substitute
ETX = end of text  VT = vertical TAB  DC3 = ESC = escape
ENT = 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. blockUS = unit separator

### UTF-8 ENCODING

<table>
<thead>
<tr>
<th>UTF-8</th>
<th>Number of bits in code point</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxxxx</td>
<td>7</td>
<td>00000000–0000007F</td>
</tr>
<tr>
<td>110xxxxx 10xxxxx</td>
<td>11</td>
<td>00000080–0000FFFF</td>
</tr>
<tr>
<td>1110xxxx 10xxxxx 10xxxxx</td>
<td>16</td>
<td>00000800–000FFFFFF</td>
</tr>
<tr>
<td>11110xxx 10xxxxx 10xxxxx 10xxxxx</td>
<td>21</td>
<td>00010000–001FFFFFFFF</td>
</tr>
<tr>
<td>111110xx 10xxxxx 10xxxxx 10xxxxx 10xxxxx</td>
<td>26</td>
<td>00200000–03FFFFFFFF</td>
</tr>
<tr>
<td>1111110x 10xxxxx 10xxxxx 10xxxxx 10xxxxx 10xxxxx</td>
<td>31</td>
<td>04000000–FFFFFFFF</td>
</tr>
</tbody>
</table>
```
25 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 25.1** ($N$ 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 25.1** ($N = 1$: 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.

<table>
<thead>
<tr>
<th>8-bit unsigned</th>
<th>All integers from 0 to 255</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit signed (two’s complement)</td>
<td>All integers from -128 to -1, 0, 1 to 127</td>
</tr>
</tbody>
</table>

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.
25.1 Unsigned Integers

Fact 25.2 (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 25.3 (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 25.2. 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 25.4 (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 rightmost bit position, or in other words by isolating the $n - k$ bits of $N$.

Example 25.3. 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.
25.2 Signed Mantissa

Fact 25.5 (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 25.4 (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.

For the $+43$ and $-43$ representation, the leftmost of the 8 bits is the sign and varies. The remaining 7 rightmost 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 representations 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 $128 + N = 2^7 + N$. 
25.3 One’s Complement

Fact 25.6 \((n\text{-bit One’s complement})\). 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 \(N \geq 0\) 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 25.5 (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 rightmost 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 | \begin{array}{c} 0 0 0 0 0 0 0 \end{array} | \text{One’s complement : positive zero} \,+0 \\
| 7 6 5 4 3 2 1 0 | \begin{array}{c} 1 1 1 1 1 1 1 \end{array} | \text{One’s complement : negative zero} \,-0 \\
| 7 6 5 4 3 2 1 0 | \begin{array}{c} 0 1 1 1 1 1 1 \end{array} | \text{One’s complement : } +127 \\
| 7 6 5 4 3 2 1 0 | \begin{array}{c} 1 0 0 0 0 0 0 \end{array} | \text{One’s complement : } -127 \\
| 7 6 5 4 3 2 1 0 | \begin{array}{c} 0 0 1 0 1 0 1 \end{array} | \text{One’s complement : } +43 \\
| 7 6 5 4 3 2 1 0 | \begin{array}{c} 1 1 0 1 0 1 0 \end{array} | \text{One’s complement : } -43
25.4 Two’s Complement

Fact 25.7 (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 representation 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 representation 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 25.6 (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.

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 $-1$ in its radix-10 representation or a sign bit of 1 in its two’s complement representation) flip and add one to the result.
25.5 Fixed-point real numbers

Fact 25.8 (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 25.7 (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 \( R = R_i + R_d = 1.00 + 0.75 = 1.75 \).

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\] 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 \( R = R_i + R_d = 3.00 + 0.50 = 3.50 \).

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\] n-bit fixed point with \( n_i = 5 \)
25.6 Floating-Point real numbers

**Definition 25.1** (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 25.9** (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 25.8.** 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 25.9.** Let \( N = 160 \) whose binary representation in \( n = 8 \) is \( 10100000 \). Then \( M = N / 2^6 \) is \( 160 \times 64 = 2.50 \). If \( N = 10100000 \) or \( 1010000. \) is shifted right 6 positions and we are left with \( 10.10000 \) in binary. (The implied decimal points is between 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 25.10** (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 \( 1.0 \), 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 exponent 2 (not the base).

**Example 25.11** (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 25.1** (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.)

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

**Definition 25.2** (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.
25.7 IEEE-754: Single Precision

Fact 25.10 (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.

Definition 25.3.

\[
\begin{array}{c|c|c}
S & E & \text{Mantissa:23} \\
\hline
1 & 8 & \\
\end{array}
\}
\quad \text{SP: 32-bit}

Fact 25.11 (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 bit \( S \). Exponents that all-0 and all-1 are reserved. The quadruplet \((S,E,F,B)\) 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 25.12 (Smallest SP value). Smallest \( E = 1 \) and then \( E - B = 1 - 127 = -126 \). The smallest fraction \( F = \text{all } -0 \), and then \( 1.F = (1+F) = 1.0 \). The smallest numbers are then \( \pm 2.0 \times 2^{-126} \).

Example 25.13 (Largest SP value). Largest \( E \) in binary is \( 1111110 \) and thus \( E = 254 \). Then \( E - B = 254 - 127 = 127 \). The largest fraction \( F = \text{all } -1 \), and then \( 1.F = (1+F) \approx 2.0 \). The largest numbers are then \( \pm 2.0 \times 2^{127} \).

Example 25.14 (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 \( F = 110 \ldots 0 \). We also have \( E = -1 + B = -1 + 127 = 126 \). The exponent \( E \) in 8-bit binary is \( E = 01111110 \).

\[
\begin{array}{c|c|c|c}
S & E & F \\
\hline
1 & 8 & \\
\end{array}
\}
\quad R = -0.75 \text{ in SP}

25.8 IEEE-754: Double Precision

Definition 25.4.

<table>
<thead>
<tr>
<th>S:1</th>
<th>Exponent:11</th>
<th>Mantissa:52</th>
</tr>
</thead>
</table>

} DP: 64-bit

Fact 25.12 (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 bit \( S \). Exponents that all-0 and all-1 are reserved. The quadruplet \((S, E, F, B)\) 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 25.15 (Smallest DP value). Smallest \( E = 1 \) and then \( E - B = 1 - 1023 = -1022 \). The smallest fraction \( F = \text{all } 0 \), and then \( 1.F = (1 + F) = 1.0 \). The smallest numbers are then \( \pm 1.0 \times 2^{-1022} \).

Example 25.16 (Largest DP value). Largest \( E \) in binary is 11111111110 and thus \( E = 2046 \). Then \( E - B = 2046 - 1023 = 1023 \). The largest fraction \( F = \text{all } 1 \), and then \( 1.F = (1 + F) \approx 2.0 \). The largest numbers are then \( \pm 2.0 \times 2^{1023} \).

Example 25.17 (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 \( F = 110 \ldots 0 \). We also have \( E = -1 + B = -1 + 1023 = 1022 \). The exponent \( E \) in 11-bit binary is \( E = 01111111110 \).

\[
6362626058557563535453253564468746544434241409835776353433231312292527252423222120591817613141132111098765432100 \}

\[
R = -0.75 \text{ in SP}
\]
Example 25.18. 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 \( D = 1.010...0 \). Converting \( D \) into decimal we get \( d = 1 + 1/4 = 1.25 \). Thus the number represented is \((1 - 2S) \times 1.25 \times 2^2 = -5.0\).

Example 25.19 (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 25.13 (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 25.1 (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.
25.9 **IEEE-754: Double Extended Precision**

**Definition 25.5 (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.

\[
\begin{array}{|c|c|}
\hline
S: & Exponent:15 \quad \text{Mantissa:64} \\
\hline
\end{array}
\]

Double Extended Precision: 80-bit

<table>
<thead>
<tr>
<th>single precision (SP)</th>
<th>32-bit</th>
<th>double precision (DP)</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias $B=127$</td>
<td></td>
<td>Bias $B=1023$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E</th>
<th>F</th>
<th>s</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>.</td>
<td>0</td>
<td>.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>.</td>
<td>0</td>
<td>.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>NotNormalized</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Reserved Values:

<table>
<thead>
<tr>
<th>E</th>
<th>F</th>
<th>s</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Smallest $E$: 0000 0001 = 1 - $B=126$

Smallest $F$: 0000 ... 0000 implies Smallest D: 1.0000 ... 0000 = 1.0 [normalized]

Largest $E$: 1111 1110 = 254 - $B=127$

Largest $F$: 1111 ... 1111 implies Largest D: 1.1111 ... 1111 ~ 2.0 [normalized]

Smallest E: 0000 0000 reserved to mean $2^{-126}$ for nonzero $F$

Smallest $F$: 0000 ... 0001 implies Smallest D: 0.0000 ... 0001 = $2^{-123}$ [unnormalized]

Largest E: 0000 0000 reserved to mean $2^{-126}$ for nonzero $F$

Largest $F$: 1111 ... 1111 implies Largest D: 0.1111 ... 1111 ~ $1-2^{-123}$ [Unnormalized]

Largest $F$: 0000 0000 implies Largest D: 0.0000 ... 0000 = $2^{-123}$ [Unnormalized]

Largest $F$: 0000 0000 implies Largest D: 0.0000 ... 0000 = $2^{-123}$ [Unnormalized]
26  Computer Architectures: von-Neumann and Harvard

26.1  Von-Neuman model of computation

Fact 26.1 (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.

26.2  Harvard model of computation

Fact 26.2 (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 26.3 (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.

26.3  CPU, Microprocessor, Chip and Die

Fact 26.4 (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 26.5 (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.
26.4 More than one

Fact 26.6 (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 26.1 (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 independently of it. In such a case a many-core system is called a coprocessor.

Fact 26.7 (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 parallel. 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. transcendental functions). Thus the effective ”core count” is in the thousands.
27 Computer Architectures: Memory Hierarchies

Fact 27.1 (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 main 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 the faster it is (low access times) but also the costlier it becomes and the scarcer/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 27.2 (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 level-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 27.3 (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 27.4 (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. \((64 + 256) \times 4KiB\) 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 only then will it go to L3. Then L3 behaves as an 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 27.5 (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 27.6 (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 27.1 (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.

<table>
<thead>
<tr>
<th>BigEndian</th>
<th>LittleEndian (Intel architecture)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10: 01010101</td>
<td>11110000</td>
</tr>
<tr>
<td>11: 11110000</td>
<td>01010101</td>
</tr>
</tbody>
</table>

Fact 27.7 (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.
28 Hard-Disk Drives (HDD)

- **Platter (or just disk)** It is a 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 outside 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 the 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.
28.1 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.

28.2 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/settling 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.

28.3 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 to be under or over the relevant head positioned under or over the active head. A 7200RPM drive completes one rotation in approximately 8.33ms.

\[
\frac{\text{Time}}{\text{Rotation}} = \frac{1\text{mn}}{7200\text{R}} = \frac{60\text{s}}{7200\text{R}} = \frac{60,000\text{ms}}{7200\text{R}} = 8.33\text{ms}/\text{Rotation} = 8.33\text{ms}/R \quad R = \text{Rotation}
\]
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 \( \frac{1}{2} \times 8.33 = 4.17 \text{ms/R} \).

### 28.4 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 area 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 than sector size worth of data, we ignore additional access, latency costs.

### 28.5 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 (e.g., 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 \( 15 + 512 + 50 = 577 \text{B} \) when it reads a (logical) sector. In other words \( 512/577 \approx 88\% \) 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).

### 28.6 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, with 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. Controller overhead is about 2ms, average seek time is roughly 8ms, and average rotational delay 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

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

28.7 Average Seek Time vs Maximum Seek Time

**Fact 28.1.** 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

\[
A = \frac{\sum_{i=0}^{N} \sum_{j=0}^{N} |i-j|}{(N+1)^2} = \frac{1}{(N+1)^2} \cdot \sum_{i=0}^{N} \sum_{j=0}^{N} |i-j| = \frac{1}{(N+1)^2} \cdot S.
\]
We compute next the sum $S$.

$$S = \sum_{i=0}^{N} \sum_{j=0}^{N} |i - j|$$

$$= \sum_{i=0}^{N} \left[ \sum_{j=0}^{i} |i - j| + \sum_{j=i+1}^{N} |i - j| \right]$$

$$= \sum_{i=0}^{N} \left[ \sum_{j=0}^{i} (i - j) + \sum_{j=i+1}^{N} (j - i) \right]$$

$$= \sum_{i=0}^{N} \left[ \sum_{j=0}^{i} i - \sum_{j=0}^{i} j + \sum_{j=i+1}^{N} j - \sum_{j=i+1}^{N} i \right]$$

$$= \sum_{i=0}^{N} \left[ i(i+1) - i(i+1)/2 + \sum_{j=0}^{N} j - \sum_{j=0}^{i} j - i(N-i) \right]$$

$$= \sum_{i=0}^{N} [N(N+1)/2 - i(N-i)]$$

$$= \sum_{i=0}^{N} N(N+1)/2 - N \cdot \sum_{i=0}^{N} i + \sum_{i=0}^{N} i^2$$

$$= N(N+1)^2/2 - N^2(N+1)/2 + N(N+1)(2N+1)/6$$

After some minor calculations we obtain the following.

$$S = \sum_{i=0}^{N} \sum_{j=0}^{N} |i - j|$$

$$= 3N(N^2 + 2N + 1) - 3N^3 - 3N^2 + 2N^3 + 3N^2 + N$$

$$= 3N^3 + 6N^2 + 3N - 3N^3 - 3N^2 + 2N^3 + 3N^2 + N$$

$$= 2N^3 + 6N^2 + 4N$$

$$= N^3 + 3N^2 + 2N$$

$$= N(N+1)(N+2)/3.$$  

Therefore from Equation 3 by replacing into Equation 3 we obtain the following.

$$A = \frac{1}{(N+1)^2} \cdot S = \frac{1}{(N+1)^2} \cdot \frac{N(N+1)(N+2)}{3} = \frac{N}{3} + \frac{N}{3(N+1)} = \frac{N + 1 - 1}{3(N+1)}$$

$$= \frac{N}{3} + \frac{1}{3(N+1)} \rightarrow \frac{N}{3} + \frac{1}{3}.$$  

□
29 Constants, Variables, Data-Types

Fact 29.1 (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 29.2 (Variable.). In computer programs we also use objects (names, aliases) whose values can change. Those objects are known as a variable.

Fact 29.3 (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 29.1. 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 29.4 (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 29.5 (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 29.6 (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 29.2 (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 29.3.** What are the primitive data types of C, C++, or Java? What are the composite data types of C, C++, or Java?

**Fact 29.7 (What is a Data Model (DM)?).** It is an abstraction that describes how data are represented and used.

**Example 29.4.** 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 29.8 (Weakly-typed and strongly-typed languages.).** In a weakly-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*
\[
x = \text{int8}(10); \quad \% z \text{ is integer (data type)}
\]
\[
x = 10.12; \quad \% z \text{ is real number (data type)}
\]
\[
x = 'abcd'; \quad \% z \text{ is a string of 4 characters (data type)}
\]

*Strongly Typed Language such as C, C++, or Java*
\[
\text{int } z; \quad \% z \text{ is a 32-bit (4B) integer whose data type can not change in the program}
\]
\[
x = 10; \quad z = 2; \quad \% \text{ ok}
\]
\[
x = 10.10; \quad \% \text{ Error or unexpected behavior: right hand-side is not an integer.}
\]
Fact 29.9 (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 \texttt{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 29.10 (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 29.11 (The ADT Dictionary.). For example a Dictionary is an abstract data type consisting of a collection of words on which a set of operations are defined such as Insert, Delete, Search.

Fact 29.12 (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 29.13 (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 29.14 (Mathematical Function: Input and Output Interface.). When we write a function such as \( f(x) = x \times x \) in Mathematics we mean that x is the unknown or parameter or indeterminate of the function. The function is defined in terms of x. The computation performed is \( x^2 \) i.e. \( x \times x \). The value 'returned' or 'computed' is exactly that \( x \times x \). 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 29.15 (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 29.16 (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 29.17 (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.
30 Formulae collection

The mathematics (not discrete mathematics) that you need can be summarized in the following one-page summary.

\[
\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \quad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \quad \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}.
\]

For \( a \neq 1 \), and \( |b| < 1 \) we have that

\[
\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}, \quad \sum_{i=0}^{n-1} i a^i = \frac{(n-1)a^{n+1} - na^n + a}{(1-a)^2},
\]

\[
\sum_{i=0}^{\infty} b^i = \frac{1}{1-b}, \quad \sum_{i=1}^{\infty} b^i = \frac{b}{1-b}, \quad \sum_{i=0}^{\infty} i b^i = \frac{b}{(1-b)^2},
\]

\[
H_n = \sum_{i=1}^{n} \frac{1}{i}, \quad \sum_{i=1}^{n} i H_i = \frac{n(n+1)}{2} - H_n - \frac{n(n-1)}{4}.
\]

\[
n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta \left(\frac{1}{n}\right)\right), \quad n! \approx \left(\frac{n}{e}\right)^n, \quad a^{\log_b a} = n^{\log_b a},
\]

\[
e \approx 2.718281, \quad \pi \approx 3.14159, \quad \gamma \approx 0.57721, \quad \phi = \frac{1+\sqrt{5}}{2} \approx 1.61803, \quad \tilde{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.61803.
\]

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

L1.

\[
\lg(ab) = \lg a + \lg b, \quad \lg(a/b) = \lg a - \lg b, \quad \lg(a^b) = b \lg a, \quad 2^{\lg(a)} = a,
\]

L2.

\[
a^x a^y = a^{x+y}, \quad a^x / a^y = a^{x-y}, \quad (a^x)^y = a^{xy}.
\]

D1.

\[
(f(x)g(x))' = f'(x)g(x) + f(x)g'(x), \quad \left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x)g(x) - f(x)g'(x)}{g^2(x)}, \quad (e^x)' = \ln(e) e^x.
\]

S1.

\[
\frac{1}{1-x} = 1 + x + x^2 + \ldots + x^j + \ldots = \sum_{i=0}^{\infty} x^i,
\]

S2.

\[
\frac{x}{(1-x)^2} = x + 2x^2 + \ldots + ix^i + \ldots = \sum_{i=0}^{\infty} ix^i,
\]

S3.

\[
e^x = 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^i}{i!} + \ldots = \sum_{i=0}^{\infty} \frac{x^i}{i!}.
\]