In [1]:

```
%%html
<style>
    p,ul {

          font-size: 18px;
          font-family: "Segoe Print","Georgia", "Verdana", "Lucida Console", "Courier N
ew";
          line-height: 150%;
    }

    .text_cell_render p {
    text-align: justify;
    text-justify: inter-word;
    }
</style>
```

In [2]:

```
from IPython.display import Image
# Image(filename = "gd5.png", width = "100%")
```

Algorithm

A finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

- **Input**. Zero or more quantities are externally supplied.
- **Output**. At least one quantity is produced.
- **Definiteness**. Each instruction is clear and unambiguous.
- **Finiteness**. If we trace out the instructions of an algorithm, then for all cases,the algorithm terminates after a finite number of steps.
- **Effectiveness**. Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.

Algorithms that are definite and effective are also called computational procedures

A program is the expression of an algorithm in a programming language

- How to devise algorithms?
  - Creating an algorithm is an art which may never be fully automated.
  - The goal is to study various design techniques that have proven to be useful in that they have often yielded good algorithms.
  - By mastering these design strategies, it will become easier for you to devise new and useful algorithms.
- How to validate algorithms?
  - To show that the algorithm computes the correct answer for all possible legal inputs.
  - The algorithm need not as yet be expressed as a program.
  - The purpose of the validation is to assure us that this algorithm will work correctly independently of the issues concerning the programming language it will eventually be written in.
  - Once the validity of the method has been shown,a program can be written and a second phase begins. This phase is referred to as program proving or sometimes as program verification.
  - A proof of correctness requires that the solution be stated in two forms.
    - One form is usually as a program which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in the predicate calculus.
    - The second form is called a specification, and this may also be expressed in the predicate calculus.
  - A proof consists of showing that these two forms are equivalent in that for every given legal input, they describe the same output. A complete proof of program correctness requires that each statement of the programming language be precisely defined and all basic operations be proved correct. All these details may cause a proof to be very much longer than the program.
- How to analyze algorithms?
  - Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage an algorithm requires. An important result of this study is that it allows you to make quantitative judgments about the value of one algorithm over another.

    - Another result is that it allows you to predict whether the software will meet any efficiency constraints that exist
- How to test a program?

  Testing a program consists of two phases: debugging and profiling (or performance measurement).
    - Debugging is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them
    - Profiling or performance measurement is the processof executing a correct program on datasets and measuring the time and space it takes to compute the results.

These timing figures are useful in that they may confirm a previously done analysis and point out logical places to perform useful optimization

Asymptotic analysis

- In mathematical analysis, asymptotic analysis, also known as asymptotics, is a method of describing limiting behavior.
- As an illustration, suppose that we are interested in the properties of a function $\mathbf{f(n)}$ as $\mathbf{n}$ becomes very large. If $\mathbf{f(n) = n^2 + 3n}$, then as $\mathbf{n}$ becomes very large, the term $\mathbf{3n}$ becomes insignificant compared to $\mathbf{n^2}$. The function $\mathbf{f(n)}$ is said to be "asymptotically equivalent to $\mathbf{n^2}$, as $\mathbf{n \to \infty}$". This is often written symbolically as $\mathbf{f(n) \sim n^2}$, which is read as "$\mathbf{f(n)}$ is asymptotic to $\mathbf{n^2}$".
- An example of an important asymptotic result is the prime number theorem. Let $\mathbf{f(x)}$ denote the prime-counting function, i.e. $\mathbf{f(x)}$ is the number of prime numbers that are less than or equal to $\mathbf{x}$. Then the theorem states that

$$\mathbf{f(x) = \frac{x}{\log(x)}}$$

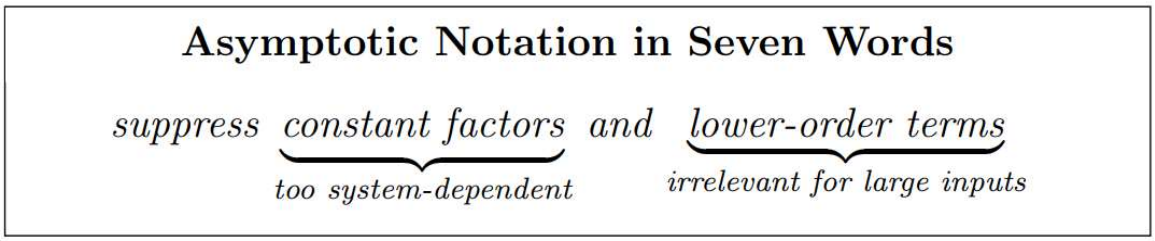
Asymptotic notation

- It provides the basic vocabulary for discussing the design and analysis of algorithms.
- Asymptotic notation is coarse enough to suppress all the details you want to ignore—details that depend on the choice of architecture, the choice of programming language, the choice of compiler, and so on.
- On the other hand, it's precise enough to make useful comparisons between different highlevel algorithmic approaches to solving a problem, especially on larger inputs (the inputs that require algorithmic ingenuity).
- Programmers often compare algorithms by saying that one algorithm runs in "big-O of n time," while another runs in "big-O of n-squared time."
- Intuitively, saying that something is $\mathcal{O}(\mathbf{f(n)})$ for a function $\mathbf{f(n)}$ means that $\mathbf{f(n)}$ is what you're left with after suppressing constant factors and lower-order terms.
- This "big-O notation" buckets algorithms into groups according to their asymptotic worst-case running times—the linear $(\mathcal{O}(\mathbf{n}))$-time algorithms, the $\mathcal{O}(\mathbf{n}\log_2\mathbf{n})$-time algorithms, the quadratic $(\mathcal{O}(\mathbf{n^2}))$-time algorithms, the constant $(\mathcal{O}(\mathbf{1}))$-time algorithms, and so on.
- We measure an algorithm's efficiency as its worst case efficiency, which is the largest amount of time an algorithm can take given the worst possible input of a given size.
- The advantage to considering the worst case efficiency of an algorithm is that we are guaranteed that our algorithm will never behave worse than our worst case estimate, so we are never surprised or disappointed. Thus, when we derive a Big-O bound, it is a bound on the worst case efficiency.

In [3]:

```
Image(filename = "an.png", width = "100%")
```

Out[3]:



Why constant factors and lower-order terms are not important in algorithm analysis?

- Lower-order terms, by definition, become increasingly irrelevant as you focus on large inputs, which are the inputs that require algorithmic ingenuity.
- Constant factors are generally highly dependent on the details of the environment. If we don't want to commit to a specific programming language, architecture, or compiler when analyzing an algorithm, it makes sense to use a formalism that does not focus on constant factors.
- When ignoring constant factors, we don't even need to specify the base of the logarithm (as different logarithmic functions differ only by a constant factor)
- even the function $\mathbf{10^{100} \cdot n}$ is technically $\mathcal{O}(\mathbf{n})$. In this course, we will only study running time bounds where the suppressed constant factor is reasonably small.

Asymptotic notation - example

The upper bound on the running time of MergeSort algorithm is

$$\mathbf{6n \log_2 n + 6n}$$

where $\mathbf{n}$ *is the length of the input array.*

- *The lower-order term here is the* $\mathbf{6n}$*, as* $\mathbf{n}$ *grows more slowly than* $\mathbf{n \log_2 n}$*, so it will be suppressed in asymptotic notation.*
- *The leading constant factor of* $\mathbf{6}$ *also gets suppressed, leaving us with the much simpler expression of* $\mathbf{n \log_2 n}$*.*
- *We would then say that the running time of MergeSort is "**big-O of n log n**," written* $\mathcal{O}(\mathbf{n \log_2 n})$*, or that MergeSort is an "**O(n log n)-time algorithm**."*

Running times of some simple algorithms

In [4]:

```
Image(filename = "safe.png", width = "100%")
```

Out[4]:

Searching One Array

Input: array A of n integers, and an integer t.
Output: Whether or not A contains t.

---

for $i := 1$ to n **do**
  if $A[i] = t$ **then**
    return TRUE
return FALSE

- In the worst case, when t is not in the array, the code will do an unsuccessful search, scanning through the entire array (over n loop iterations) and returning false.
- The key observation is that the code performs a constant number of operations for each entry of the array (comparing A[i] with t, incrementing the loop index i, etc.). Here "constant" means some number independent of n, like 2 or 3. We could argue about exactly what this constant is in the code above, but whatever it is, it is conveniently suppressed in the big-O notation.
- Similarly, the code does a constant number of operations before the loop begins and after it ends, and whatever the exact constant may be, it constitutes a lower-order term that is suppressed in the big-O notation.
- Since ignoring constant factors and lower-order terms leaves us with a bound of n on the total number of operations, the asymptotic running time of this code is O(n).
- Equivalently, we say that the algorithm has running time linear in n.

In [5]:

```
Image(filename = "sasfe.png", width = "100%")
```

Out[5]:

Searching Two Arrays

Input: arrays A and B of n integers each, and an integer t.
Output: Whether or not A or B contains t.

---

for $i := 1$ to n **do**
  if $A[i] = t$ **then**
    return TRUE
for $i := 1$ to n **do**
  if $B[i] = t$ **then**
    return TRUE
return FALSE

- First we search the first array, and then the second array.
- This extra factor of 2 contributes only to the leading constant in the running time bound and is therefore suppressed when we use big-O notation.
- So this algorithm, like the previous one, is a linear-time algorithm.

In [6]:

```
Image(filename = "sasfce.png", width = "100%")
```

Out[6]:

Checking for a Common Element

Input: arrays A and B of n integers each.
Output: Whether or not there is an integer t contained in both A and B.

---

for $i := 1$ to n **do**
  for $j := 1$ to n **do**
    if $A[i] = B[j]$ **then**
      return TRUE
return FALSE

- Here, for each of the $\mathbf{n}$ iterations of the outer for loop, the code performs $\mathbf{n}$ iterations of the inner for loop.
- This gives $\mathbf{n} \times \mathbf{n} = \mathbf{n^2}$ iterations in all.
- This algorithm has the running time of $\mathcal{O}(\mathbf{n^2})$. ("Big-O of n squared," also called a "quadratic-time algorithm.")
- So with this algorithm, if you multiply the lengths of the input arrays by 10, the running time will go up by a factor of 100 (rather than a factor of 10 for a linear-time algorithm)

In [7]:

```
Image(filename = "safds.png", width = "100%")
```

Out[7]:

Checking for Duplicates

Input: array A of n integers.
Output: Whether or not A contains an integer more than once.

---

for $i := 1$ to n **do**
  for $j := i + 1$ to n **do**
    if $A[i] = A[j]$ **then**
      return TRUE
return FALSE

There is exactly one iteration for each subset $\{\mathbf{i},\mathbf{j}\}$ *of two distinct indices from* $\{\mathbf{1},\mathbf{2},\ldots,\mathbf{n}\}$*, and there are precisely* $\binom{n}{2} = \frac{n(n-1)}{2}$ *such subsets.*

Big-$\mathcal{O}$ notation

In [8]:

```
Image(filename = "bon2.png", width = "100%")
```

Out[8]:

Big-O Notation (English Version)

$T(n) = O(f(n))$ if and only if $T(n)$ is eventually bounded above by a constant multiple of $f(n)$.

In [9]:

```
Image(filename = "bon3.png", width = "100%")
```

Out[9]:

Big-O Notation (Mathematical Version)

$T(n) = O(f(n))$ if and only if there exist positive constants c and n_0 such that

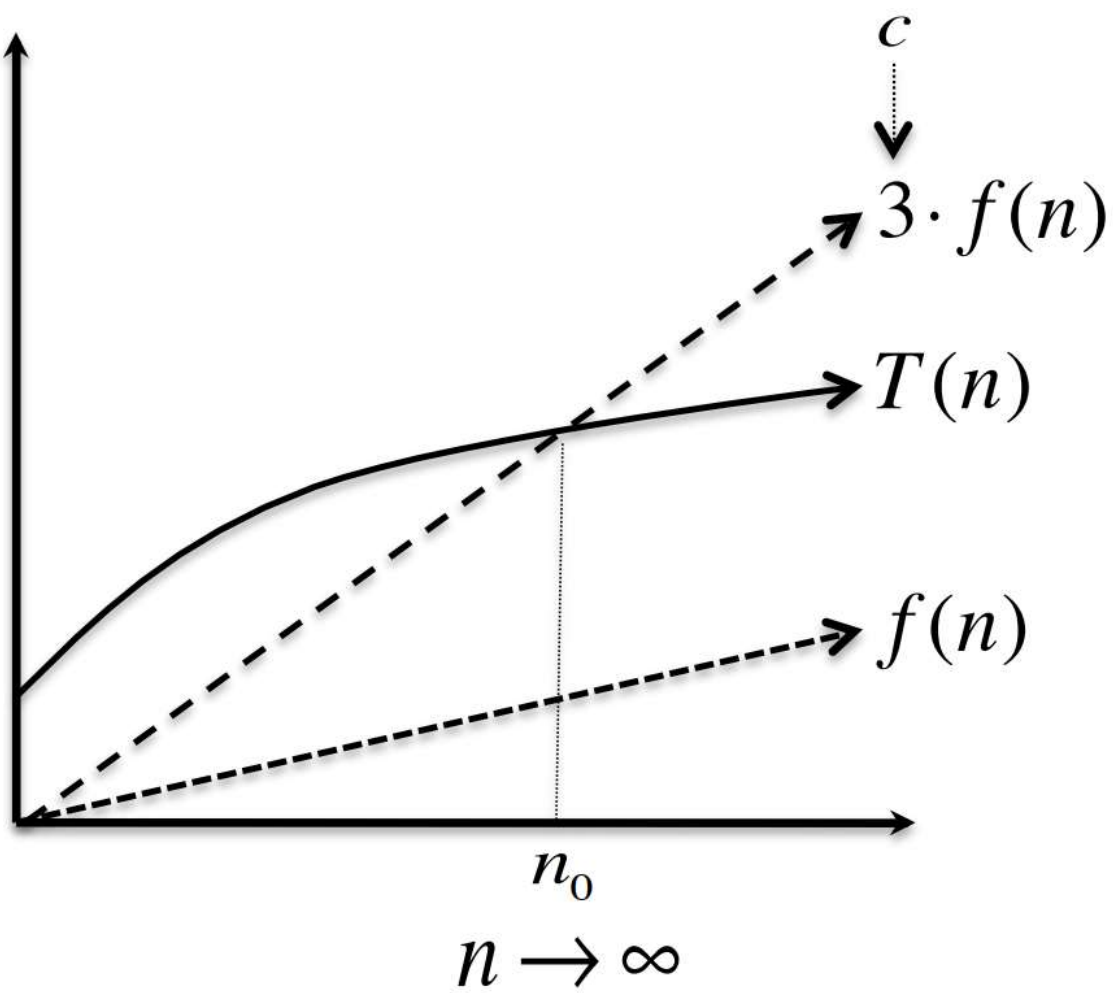
$$T(n) \leq c \cdot f(n) \tag{2.1}$$

for all $n \geq n_0$.

In [10]:

```
Image(filename = "bon4.png", width = "100%")
```

Out[10]:



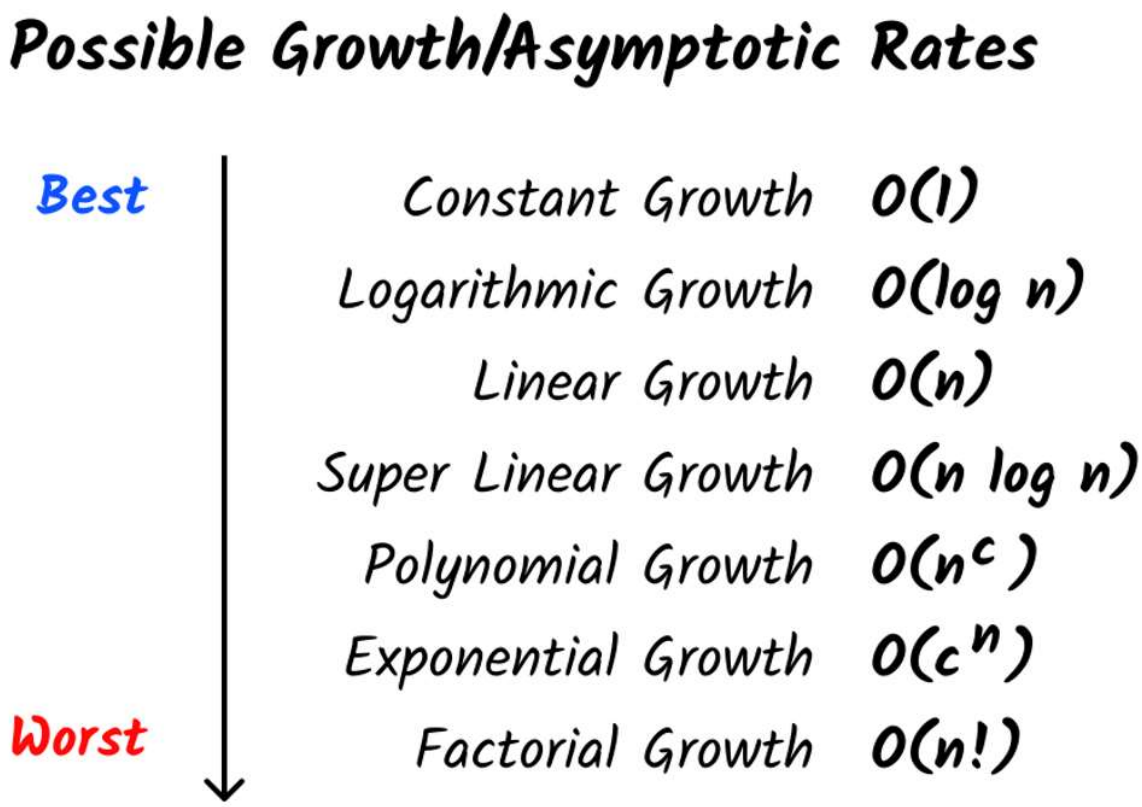
Complexity Classes

- *If the complexity of an algorithm is $\mathcal{O}(\mathbf{n}^{\mathbf{c}})$ for some constant $\mathbf{c}$, it is said to be polynomial in $\mathbf{n}$.*
- *A problem for which a polynomial algorithm exists is said to be polynomial, and the class of such problems bears the name $\mathbf{P}$.*
- *Unhappily, not all problems are polynomial, and numerous problems exist for which no polynomial algorithm has been found to this day.*

In [11]:

```
Image(filename = "an3.png", width = "100%")
```

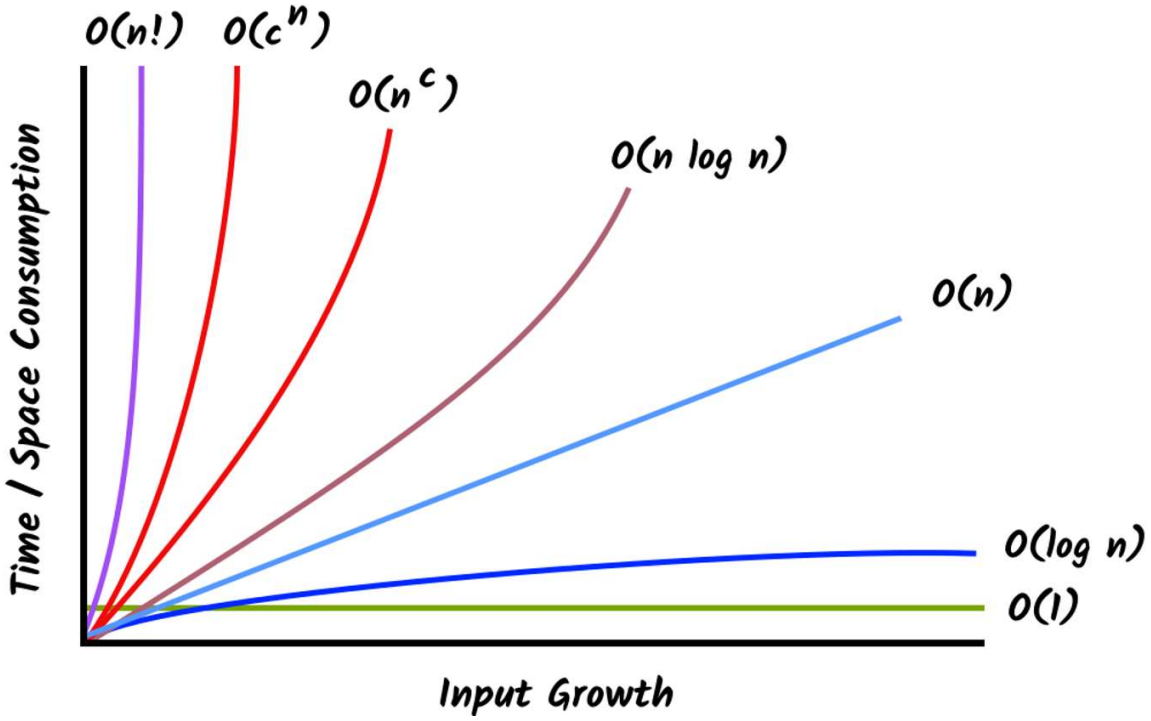
Out[11]:



In [12]:

```
Image(filename = "bon1.png", width = "100%")
```

Out[12]:



- Suppose algorithm A and B have running time of O(n) O(n) and O(n^2) respectively. For sufficiently large input sizes, algorithm A will run faster than algorithm B. That does not mean that algorithm A will always run faster than algorithm B.
- Let's consider an algorithm whose running time is given by $\mathbf{f(n) = 3n^3 + 4n + 2}$. Let us try to verify that this algorithm's time complexity is in $\mathbf{O(n^3)}$. To do this, we need to find a positive constant $\mathbf{c}$ and an integer $\mathbf{n_0 \geq 0}$, such that for all $\mathbf{n \geq n_0}$, $3n^3 + 4n + 2 \leq c \cdot n^3$

Big Omega notation

In [13]:

```
Image(filename = "big_omega.png", width = "100%")
```

Out[13]:

Big-Omega Notation (Mathematical Version)

$T(n) = \Omega(f(n))$ if and only if there exist positive constants c and n_0 such that

$$T(n) \geq c \cdot f(n)$$

for all $n \geq n_0$.

Big Theta notation

In [14]:

```
Image(filename = "big_theta.png", width = "100%")
```

Out[14]:

Big-Theta Notation (Mathematical Version)

$T(n) = \Theta(f(n))$ if and only if there exist positive constants c_1, c_2, and n_0 such that

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$$

for all $n \geq n_0$.

Little-o Notation

$T(n) = o(f(n))$ if and only if for every positive constant $c > 0$, there exists n_0 such that

$$T(n) < c \cdot f(n)$$

for all $n \geq n_0$

- Proving that one function is big-O of another requires only two constants c and n_0, chosen once and for all.
- To prove that one function is little-o of another, we have to prove something stronger, that for every constant c, no matter how small, T(n) is eventually bounded above by the constant multiple $c \cdot f(n)$.
- Note that the constant n_0 depend on c (but not on n), with smaller constants c generally requiring bigger constants n_0.
- For example, for every positive integer k, $n^{k-1} = o(n^k)$.