(To avoid cluttering, `n >= 1` assumed.)

- ***Constant*** time version:

```
unsigned long calcSumO1(unsigned long n) // n >= 1 assumed
{
   return n * (n + 1) / 2;
}
```

- ***Logarithmic*** time version:

```
unsigned long calcSumOln(unsigned long n) // n >= 1 assumed
{
   unsigned long i, sum = 0;
   for (i = 1; 2*i < n; i *= 2)
          sum += (3*i - 1)*i / 2;
   sum += (i + n) * (n - i + 1) / 2;

   return sum;
}
```

- ***Linear*** time version:

```
unsigned long calcSumOn(unsigned long n) // n >= 1 assumed
{
   unsigned long sum = 0;
   for (unsigned long i = 1; i <= n; ++i)
      sum += i;
   return sum;
}
```

- ***Linear-logarithmic*** time version:

```
unsigned long calcSumOnln(unsigned long n) // n >= 1 assumed
{
   unsigned long sum = 0, i, numIter;
   for (i = 1, numIter = 0; 2*i < n; i *= 2, ++numIter)
      for (unsigned long j = 1; j <= n; ++j)
         sum += j;
   sum /= numIter;
   return sum;
}
```

- ***Quadratic*** time version:

```
unsigned long calcSumOn2(unsigned long n) // n >= 1 assumed
{
   unsigned long sum = 0;
   for (unsigned long i = 1; i <= n; ++i)
      for (unsigned long j = 1; j <= i; ++j)
         ++sum;
   return sum;
}
```

- Big-O portrays, in broad and blended terms, an ***upper-bound order*** for the ***growth rate*** (*i.e.*, a *"growth-rate-trend"* rank of the *"no-worse-than"* type) of the *resource requirements* of an algorithm as *input size* becomes *sufficiently large*.

  (Note that it involves ***growth rate***, not ***amount***, and that *resource requirements* are typically time or space requirements.)

  - It gives *relative* information about the algorithm's resource requirements.
    - Information about how quickly (in *what kind of order/trend*, not *by how much*) resource requirements will grow as input size increases.
    - Information useful when wanting to know about the *scalability* of the algorithm: How well will it scale as input size increases? Will resource requirements quickly become sky-high as to make the algorithm not viable/practical?
  - It does not give *absolute* information about the algorithm's resource requirements.
    - Information such as the specific *amount* of resource needed for a certain input size (or *by what specific amount* the needed resource will increase when input size increases from *x* to *y*).
  - It is meant for capturing *"long-term" ("settled-down")* behavior, not *"short-term" ("settling-in")* behavior.
    - (which is the reason for "as *input size* becomes *sufficiently large*" and often abstractly described as *asymptotically*)
  - It is a "broad-brush" tool by design.
    - Its definition blends away effects due to *constant factors* and *all but the most dominant* contributors.

- Most not-so-mathematically/theoretically-oriented people would probably prefer an ***upper-bound order*** (on growth rate) specified in terms of something that the *actual growth rate* of the algorithm *will not exceed*.
  - But that is *not* (in general) what is specified in conventional Big-O notation.
    - What is usually specified is an *order* (of growth rate) in *canonical form* (= reduced to the simplest and most significant form possible without loss of generality).
      - As mentioned above, effects due to constant factors and *all but the most dominant* contributors are blended away.
      - ***E.g.***: one would say $R(n) = 9n^2 + 5n$ is $O(n^2)$, not is $O(9n^2)$ or is $O(n^2 + n)$ or is $O(9n^2 + 5n)$.
    - For starters, it is perhaps useful to think of the order specified in conventional Big-O notation as one that, when *suitably multiplied by some convenient constant factor*, will give a growth rate the *actual growth rate* of the algorithm *will not exceed*.
      - ***Check***: The *actual growth rate* of an $O(n^2)$ algorithm *can exceed that of* n^2, true or false?
  - Furthermore, the (mathematical) definition of Big-O sets no requirement on how tightly (closely) the specified growth rate order/trend tracks the actual growth rate order/trend. (*I.e.*, the definition of Big-O does *not* include any tightness requirement.)
    - The specified growth rate order/trend can be any of those that (upon *constant-factor-multiplication* where necessary) are "*as fast as or faster than*" the actual growth rate order/trend.
    - ***E.g.***: $R(n) = 9n^2 + 5n$ is $O(n^2)$, $R(n) = 9n^2 + 5n$ is $O(n^3)$, $R(n) = 9n^2 + 5n$ is $O(n^4)$, *etc.* are all mathematically correct.
    - In practice, however, a growth rate order/trend that is *as tight as possible* is desirable since it is the *most useful*.
      - Think of the difference in usefulness (to a recruiter) when a *1.5-GPA* student is categorized as *no better than 4.0-GPA* or *no better than 3.0-GPA* instead of *no better than 2.0-GPA*.
    - When analyzing an algorithm to characterize/categorize it in the *most useful* way using Big-O, we'd therefore want to arrive at a growth rate order/trend that is *as tight as possible*.
    - A common approach toward this goal is to analyze the algorithm's resource requirements in the *worst-case* scenario.
      - The hope is that the result of such analysis (perhaps a *growth function* that describes how resource requirements grow with input size) can provide revealing/supporting information for the characterizing-as-tightly-as-possible exercise/outcome.
      - The approach is undoubtedly useful for characterizing/categorizing very simple algorithms (such as those typically used to provide examples when introducing students to algorithm analysis.)
      - Unfortunately, it tends to breed in some the ***misconception*** (in general) that the growth rate order specified in conventional Big-O notation gives the algorithm's *actual growth rate* in the *worst-case scenario*.
      - ***Check***: The *actual growth rate* of an $O(n^2)$ algorithm *in the worst case* is that of n^2, true or false?