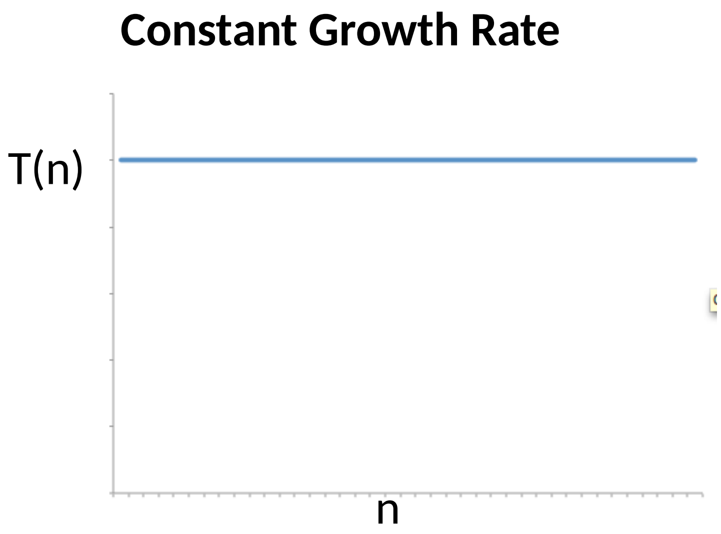
Data Structures and Algorithms

Algorithm Analysis

- Growth rates
- Big O
- Big Teta
- Big Omega

Growth rates

- Algorithms analysis is all about understanding growth rates.
- That is as the amount of data gets bigger, how much more resource will my algorithm require?
- Typically, we describe the resource growth rate of a algorithms in terms of a function.
- Some common growth rates: constant, linear, quadratic, cubic, exponential

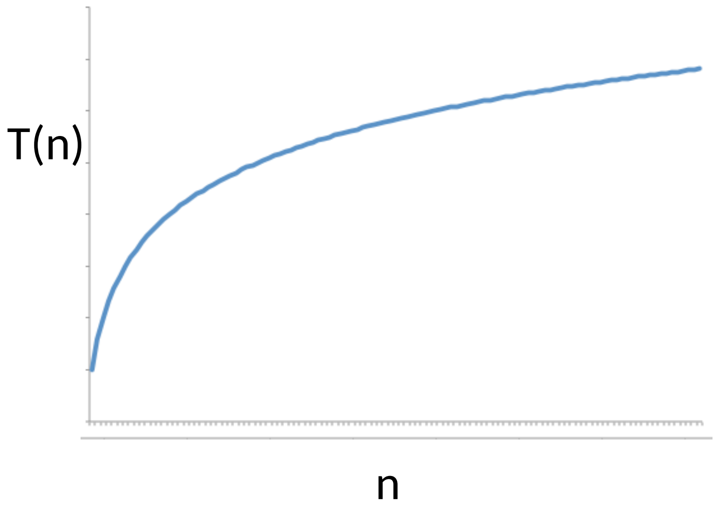


A constant resource need is one where the resource need does not grow.
That is processing 1 piece of data takes the same amount of resource as processing 1 million pieces of data.
The graph of such a growth rate looks like a horizontal line

Growth rates

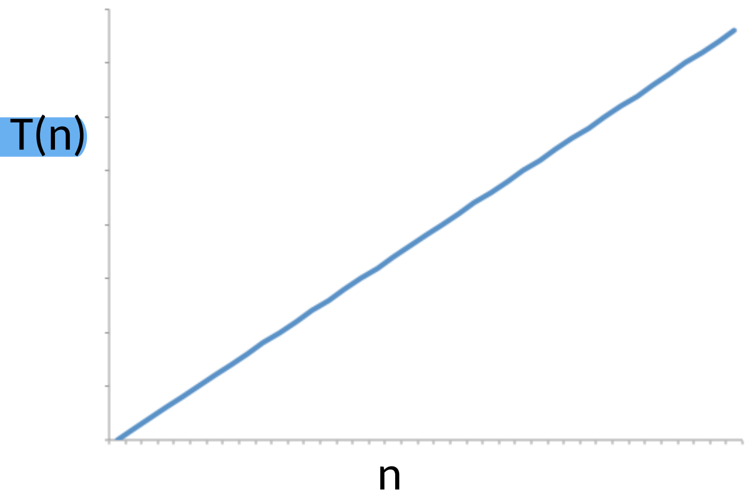
- **Linear Growth Rate**

- A linear growth rate is a growth rate where the resource needs and the amount of data is directly proportional to each other. That is the growth rate can be described as a straight line that is not horizontal.



- **Logrithmic Growth Rate**

- A logarithmic growth rate is a growth rate where the resource needs grows by one unit each time the data is doubled.
- This effectively means that as the amount of data gets bigger, the curve describing the growth rate gets flatter (closer to horizontal but never reaching it).



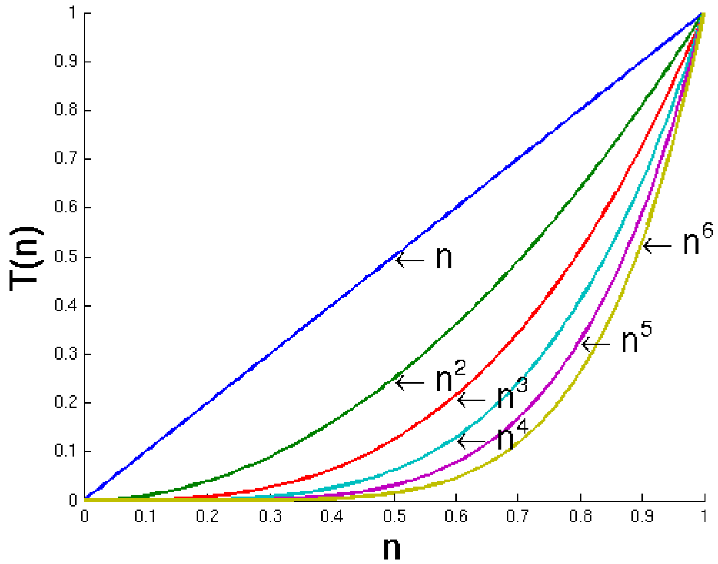
Growth rates

- **Quadratic Growth Rate**

- A quadratic growth rate is one that can be described by a parabola.

- **Cubic Growth Rate**

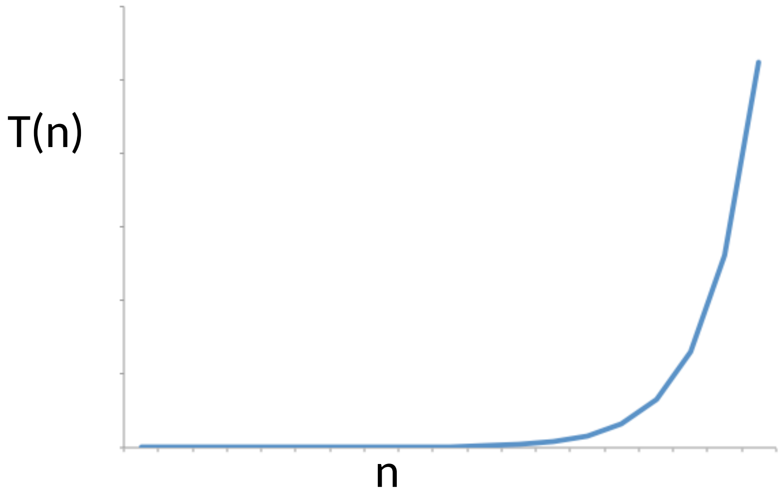
- While this may look very similar to the quadratic curve, it grows significantly faster



https://english.stackexchange.com/questions/267264/how-do-you-explain-cubic-growth-of-a-function

- **Exponential Growth Rate**

- An exponential growth rate is one where each extra unit of data requires a doubling of resource.
- As you can see the growth rate starts off looking like it is flat but quickly shoots up to near vertical (note that it can't actually be vertical)



Asymptotic notations

- Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

- For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

- But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

- When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

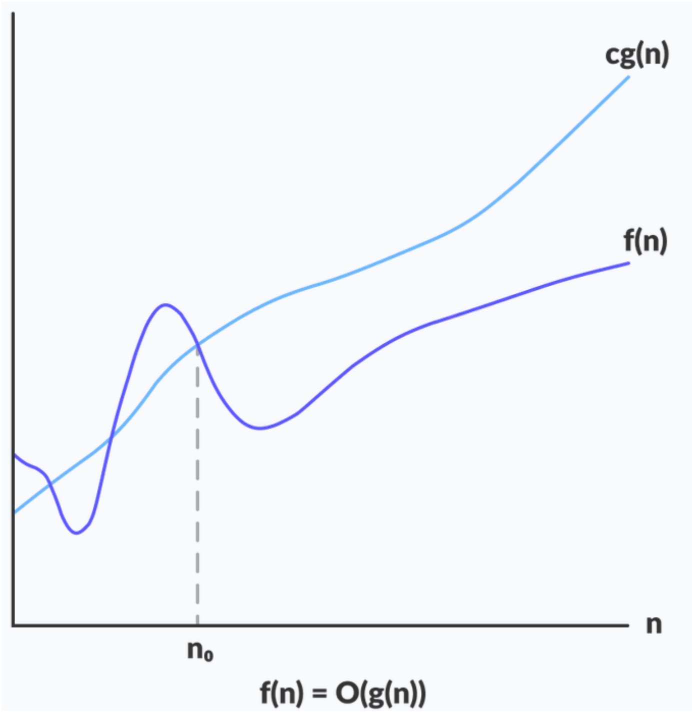
- There are mainly three asymptotic notations:
- big-O notation
- big-Θ notation.
- big-Ω notation.

Big O

- Big-O notation represents the upper bound of the running time of an algorithm.
- Thus, it gives the worst-case complexity of an algorithm.

O(g(n)) = { f(n): there exist positive constants c and n0 such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }

- The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.
- For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).
- Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.



https://www.programiz.com/dsa/asymptotic-notations

Big O

- $T(n) = O(n^3)$, which is identical to $T(n) \in O(n^3)$
- $T(n)$ grows asymptotically no faster than n^3

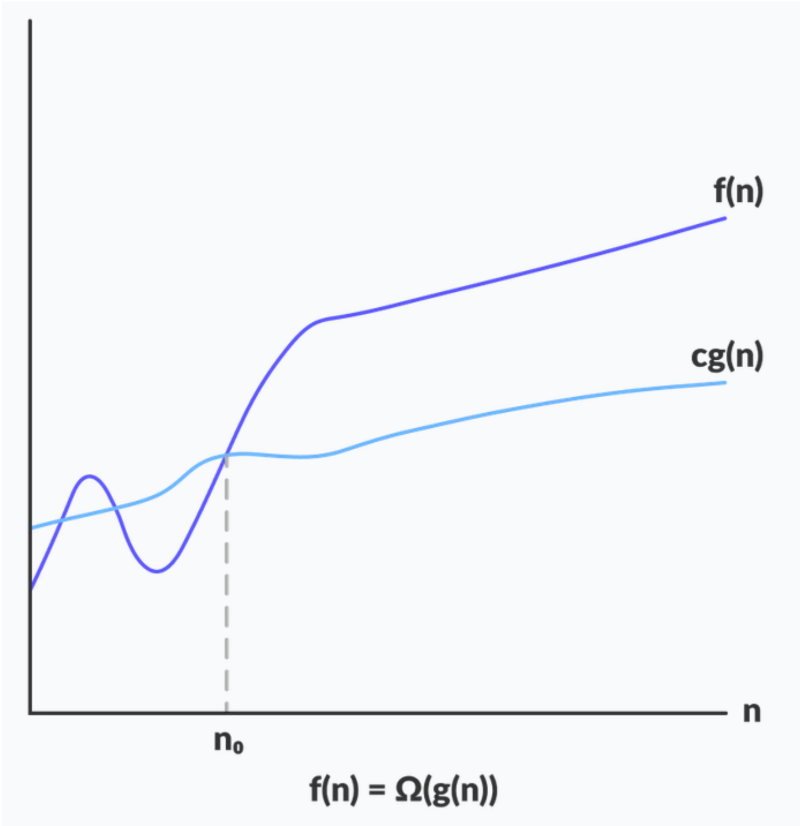
- $T(n) = \Theta(n^3)$, which is identical to $T(n) \in \Theta(n^3)$
- $T(n)$ grows asymptotically as fast as n^3

https://en.wikipedia.org/wiki/Big_O_notation

Big Omega

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n0 \}$

- Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.
- The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.
- For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

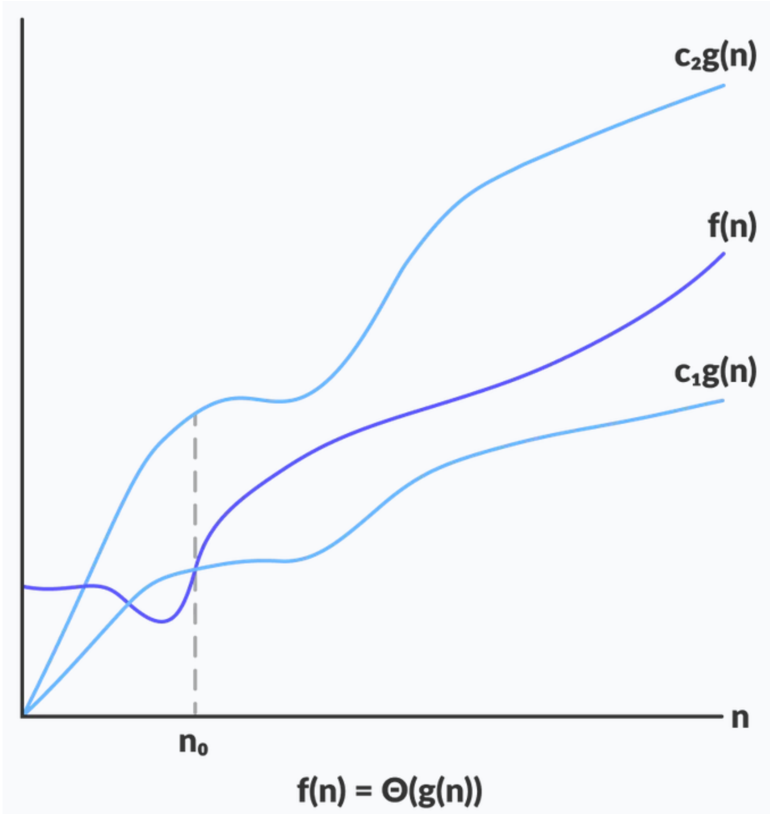


https://www.programiz.com/dsa/asymptotic-notations

Big Teta

- Theta notation encloses the function from above and below.
- Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.
- The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n), for sufficiently large n.
- If a function f(n) lies anywhere in between c1g(n) and c2g(n) for all n ≥ n0, then f(n) is said to be asymptotically tight bound.

Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0 such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0



https://www.programiz.com/dsa/asymptotic-notations

Comparison

	Big Oh	Big Omega	Big Theta
1.	It is like <= rate of growth of an algorithm is less than or equal to a specific value	It is like >= rate of growth is greater than or equal to a specified value	It is like == meaning the rate of growth is equal to a specified value
2.	The upper bound of algorithm is represented by Big O notation. Only the above function is bounded by Big O. asymptotic upper bond is it given by Big O notation.	The algorithm's lower bound is represented by Omega notation. The asymptotic lower bond is given by Omega notation	The bounding of function from above and below is represented by theta notation. The exact asymptotic behavior is done by this theta notation.
3.	Big oh (O) – Worst case	Big Omega (Ω) – Best case	Big Theta (Θ) – Average case
4.	Big-O is a measure of the longest amount of time it could possibly take for the algorithm to complete.	Big- Ω is take a small amount of time as compare to Big-O it could possibly take for the algorithm to complete.	Big- Θ is take very short amount of time as compare to Big-O and Big-? it could possibly take for the algorithm to complete.
5.	Mathematically – Big Oh is 0 <=f(n) <= c g(n) for all n>=n0	Mathematically – Big Omega is O<= C g(n) <= f(n) for all n>=n 0	Mathematically – Big Theta is O<=C 2 g(n)<=f(n)<=C 1 g(n) for n>=n 0

Example about guessing running times

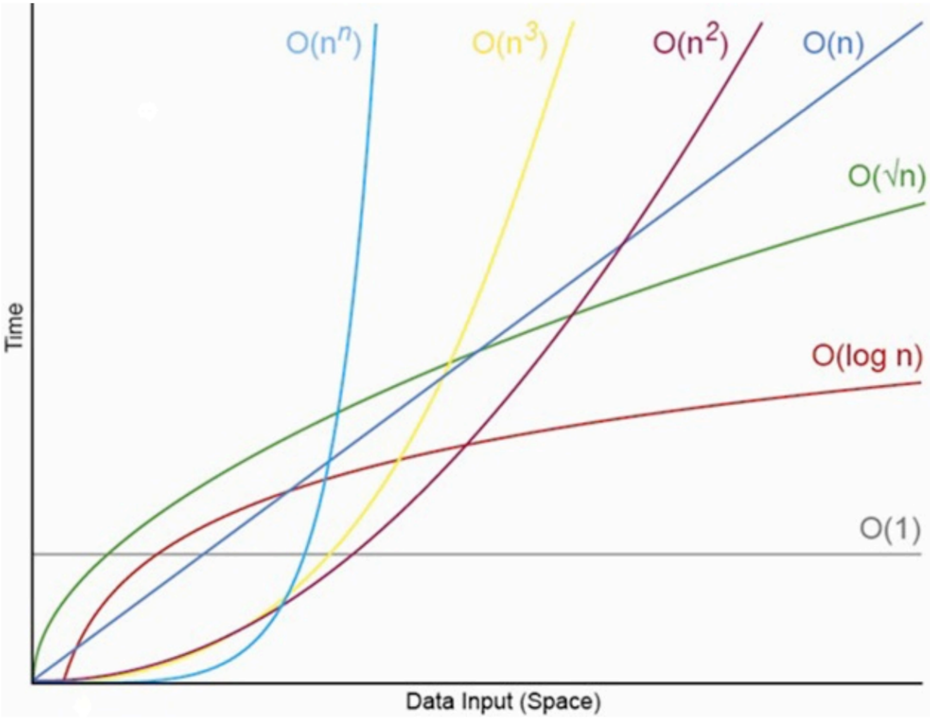
- If a program takes 10ms to process one item, how long will it take for 1000 items?
- (time for 1 item) x (Big-O( ) time complexity of N items)

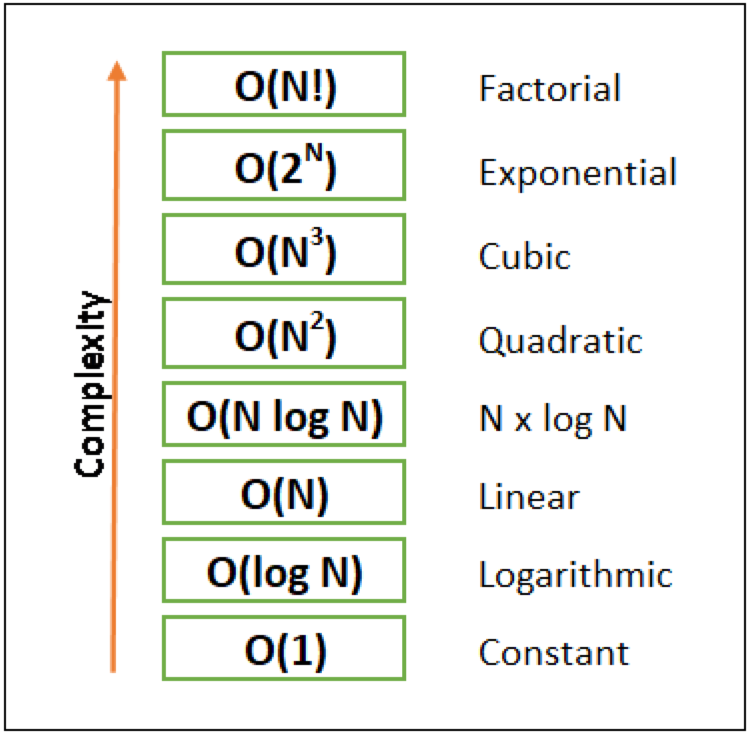
$\log_{10} N$	3 x 10ms	.03 sec
N	10^3 x 10ms	10 sec
$N \log_{10} N$	10^3 x 3 x 10ms	30 sec
N^2	10^6 x 10ms	16 min
N^3	10^9 x 10ms	12 days

Example growth rates

Big-O Characterization		Example
O(1)	*constant*	Adding to the front of a linked list
O(log N)	*log*	Binary search
O(N)	*linear*	Linear search
O(N log N)	*n-log-n*	Binary merge sort
O(N^2)	*quadratic*	Bubble Sort
O(N^3)	*cubic*	Simultaneous linear equations
O(2^N)	*exponential*	The Towers of Hanoi problem

Example growth rates





Examples

executed *n* times

```
for (i = 1; i <= n; i++)
{
    m = m + 2 ;
}
```

constant time

Runnig time :

$$T(n) = cn = \mathbf{O(\boldsymbol{N})}$$

Example

outer loop executed n times

```
for (i = 1; i <= n; i++)
{
  for (j = 1; j <= n; j++)
  {
    k = k+1 ;
  }
}
```

inner loop executed n times

constant time

$T(n) = c * n * n * = cn^2 =$ **$O(N^2)$**

Example

executed
n times

```
for (i = 1; i <= n; i++) {
  for (j = 1; j <= i; j++) {
    k = k + i + j;
  }
}
```

inner loop
executed
i times

constant time

Time Complexity

$$T(n) = c + 2c + 3c + 4c + \ldots + nc = cn(n+1)/2 = (c/2)n^2 + (c/2)n = O(n^2)$$

Ignore multiplicative constants

Ignore non-dominating terms

Example

executed
n times

```
for (i = 1; i <= n; i++) {
   for (j = 1; j <= 20; j++)
{
      k = k + i + j;
   }
}
```

inner loop
executed
20 times

constant time

Time Complexity

$$T(n) = 20 * c * n = O(n)$$

Example

```
for (j = 1; j <= 10; j++)
{
  k = k + 4;
}
for (i = 1; i <= n; i++) {
   for (j = 1; j <= 20; j++)

   {
     k = k + i + j;
    }
}
```

executed
10 times

executed
n times

inner loop
executed
20 times

Time Complexity

$$T(n) = c * 10 + 20 * c * n = O(n)$$

Example

constant time (c_0)

```
x = x +1;
for (i=1; i<=n; i++)
{
  m = m + 2;
}
for (i=1; i<=n; i++)
{
   for (j=1; j<=n; j++)
   {
      k = k+1;
   }
}
```

constant time (c_1)

executed
n times

outer loop
executed
n times

inner loop
executed
n times

constant time (c_2)

Total time = $c_0 + c_1 n + c_2 n^2 = \mathbf{O(}\boldsymbol{N}^{\mathbf{2}}\mathbf{)}$

Example

```
for (i=1; i<n; i++) {
    a[i] = 0;
}
for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i] = a[i] + i + j;
    }
}
```

$O(1)$ $O(n)$

$O(1)$ $O(n)$ $O(n^2)$

$O(n^2)$

Example

- $f1(n) = 10\ n + 25\ n^2$
- $f2(n) = 20\ n \log n + 5\ n$
- $f3(n) = 12\ n \log n + 0.05\ n^2$
- $f4(n) = n^{1/2} + 3\ n \log n$

- $O(n^2)$
- $O(n \log n)$
- $O(n^2)$
- $O(n \log n)$