Chapter 1:
Introduction to Algorithms

Contents

Algorithms

The word ‘Algorithm’ comes from the name of a Persian author, Abu Ja’ar Mohammed ibn Musa al Khwarizmi (c. 825 A.D.), Latinized Algoritmi.

He wrote an Arabic language treatise on the Hindu–Arabic numeral system, which was translated into Latin during the 12th century under the title *Algoritmi de numero Indorum* (meaning “Algoritmi on the numbers of the Indians”).

The Latin name *algoritmi* was altered to *algorismus*, *algorithmus*, *algorithme* (in French), and finally to *algorithm* (in English).

Algorithms

- Any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- All algorithms must satisfy the following criteria:
    1. **Input**: Zero or more externally supplied quantities
    2. **Output**: Produce at least one quantity
    3. **Definiteness**: Clear and unambiguous instructions
    4. **Finiteness**: Terminate after a finite number of steps
    5. **Effectiveness**: Feasible instructions

Algorithms

- Any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- All algorithms must satisfy the following criteria:
  1. **Input**: Zero or more externally supplied quantities
  2. **Output**: Produce at least one quantity
  3. **Definiteness**: Clear and unambiguous instructions
  4. **Finiteness**: Terminate after a finite number of steps
  5. **Effectiveness**: Feasible instructions
- Algorithm vs program:
  Program = The expression of an algorithm in a programming language
  Program does not need to satisfy criterion #4.

Algorithms

Study of algorithms includes

1. How to devise algorithms
   a. Different techniques/design strategies, e.g. divide and conquer, branch and bound, dynamic programming etc.
2. How to validate algorithms
   a. Show that the algorithm computes the correct answer for all possible legal inputs
   b. 2 phases:
      i. Algorithm validation
      ii. Program proving or program verification

Algorithms

3. How to analyse algorithms
   a. Performance analysis
   b. Determining the time and storage needs of an algorithm
   c. To make quantitative judgements about the value of one algorithm over another
   d. To predict whether the software will meet any efficiency constraints that exist
4. How to test a program
   a. Debugging: Executing programs on sample data to determine whether faulty results occur
   b. Profiling: Executing a correct program on data sets and measuring the time and space it takes to compute the results

Algorithm Specification

Algorithms can be described in many ways.

1. Using a natural language like English
2. Using flow charts
3. Using pseudocodes

Example

Devise an algorithm to find the largest element in an array.

Example

Algorithm: Find the largest element in an array
Input: Array A of size n
Output: The maximum element in A

Steps:

```
1. Set Result to the first element of A
2. Set i to 1
3. If A[i] is greater than Result, then set Result to A[i]
4. Increase i by 1
5. If i is less than n, go to Step 3
6. Return Result
```

Example

Algorithm: Find the largest element in an array
Input: Array A of size n
Output: The maximum element in A

Steps:

```
1.  Result ← A[0]
2.  i ← 1
3.  Repeat
      a.  If A[i] > Result
            i.  Result ← A[i]
      b.  Endif
4.  Until i >= n
5.  Return Result
```

Example

Algorithm: Find the largest element in an array
Input: Array A of size n
Output: The maximum element in A

Steps:

```
1. Result ← A[0]
2. For i ← 1 to n-1 do
    a. If A[i] > Result
        i. Result ← A[i]
    b. Endif
3. Endfor
4. Return Result
```

Pseudocode conventions

- // or # for comments
- Braces { } for blocks such as a collection of simple statements
- Assignments of values to variables using =, ≔ or ⟵
- Loops:

```
while <condition> do
{
    <statement 1>
         ⋮
    <statement 2>
}
```

Pseudocode conventions

- Loops:

```
for  variable = value$_1$ to value$_n$ step <step>  do
{
    <statement 1>
          ⋮
    <statement 2>
}
```

Pseudocode conventions

- Loops:

```
repeat
    <statement 1>
          ⋮
    <statement 2>
until <condition>
```

Example

```
Algorithm Max(A, n)
// A is an array of size n
{
        Result = A[0];
        for i = 1 to n - 1 do
                if A[i] > Result then Result = A[i];
        return Result;
}
```

Exercise

1. Devise an algorithm to add two matrices, A and B.

Why analyze an algorithm?

- Classify problems and algorithms by difficulty
- Predict performance, compare algorithms, tune parameters
- Better understand and improve implementations and algorithms
- Intellectual challenge

Sedgewick, R., & Wayne, K. (2011). Algorithms. Addison-wesley professional.

Performance analysis

Two main resources to consider when writing a program:

1. Memory
2. Time

Performance analysis focuses on obtaining **machine-independent estimates** of time and space

Space complexity

The amount of memory the algorithm needs to run to completion, S(P)
= fixed space requirements, c + variable space requirements, $S_p(I)$

Fixed space requirements, c, include space needed to store the code, simple variables, fixed-sized structured variables, and constants.

Variable space requirements, $S_p(I)$, include

- Space needed by structured variables whose size depends on the particular instance I of the program
- Additional space required when a function uses recursion.

Time complexity

The amount of computer time the algorithm needs to run to completion
= compile time + execution time T_P

Compile time does not depend on the instance characteristics. So, we are concerned only with the program's execution time, T_P

Instance characteristics include the number, size, and values of the inputs and outputs associated with I.

For example, if the input is an array containing n numbers, then n is an instance characteristic.

Time complexity

The more instructions the program contains, the more time it will take to complete.

Counting the number of operations the program performs, we can get a machine-independent estimate of its execution time.

Program step:
A syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics (i.e. the number, size, and values of inputs and outputs associated with a program instance)

Example

Count program steps

```
float sum(float a, float b) {
{
      return a+b;
}
```

Example

Count program steps

```
1.  Algorithm Sum(A, B, m, n) {
2.  // Algorithm to add two matrices
    // A and B of size m x n.
3.  {
4.      C = A
5.      for i = 0 to m-1 do
6.          for j = 0 to n-1 do
7.              C[i,j] += B[i,j]
8.      return C
9.  }
```

Order of growth

Algorithm analysis is about understanding the growth in resource consumption as the amount of data increases

As the amount of data gets bigger, how much more resource will my algorithm require?

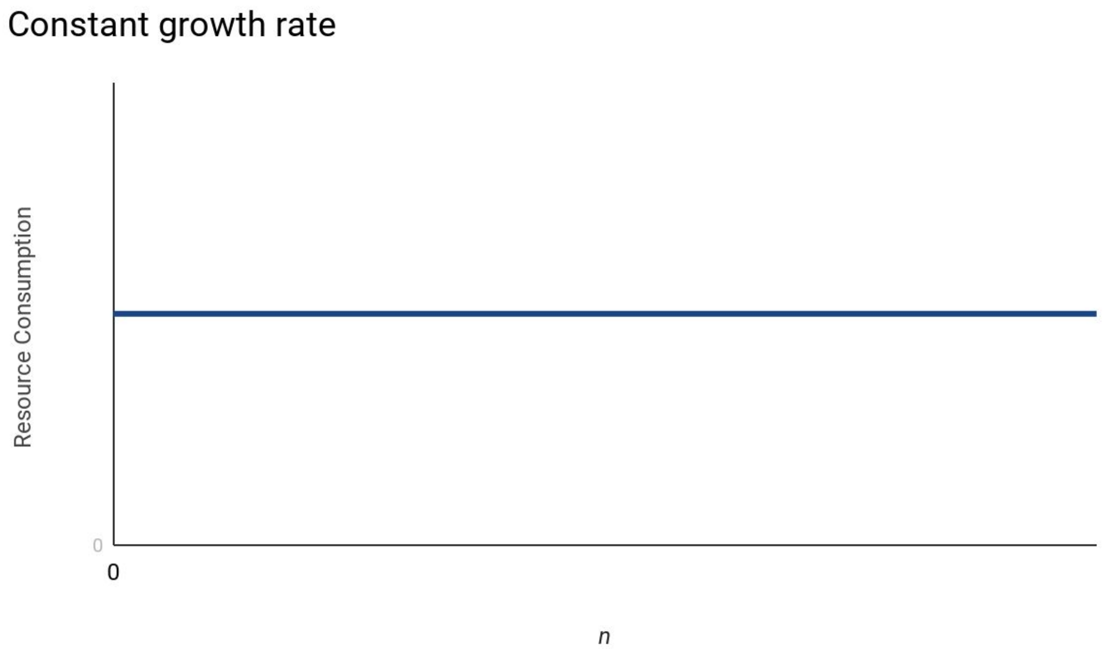
The order of growth of the running time of an algorithm

1. Gives a simple characterization of the **algorithm's efficiency**
2. Allows us to **compare the relative performance** of alternative algorithms

Constant growth rate

The resource need does not grow.

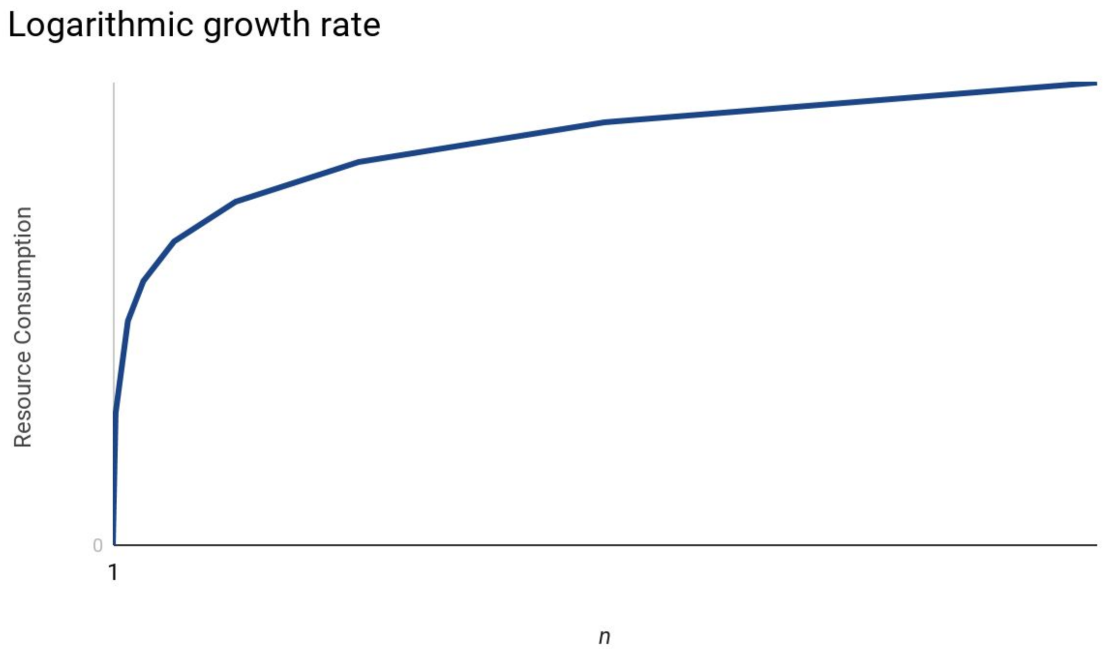
Processing 1 piece of data takes the same amount of resource as processing 1 million pieces of data



Logarithmic growth rate

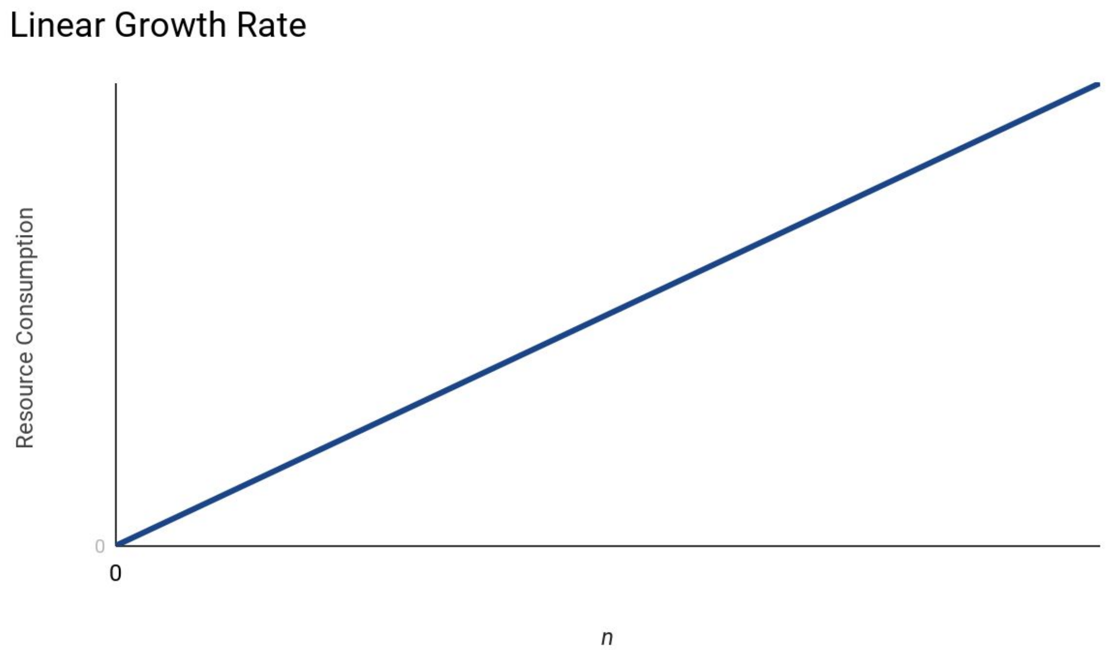
The resource needs grow by one unit each time the data is doubled.

As the amount of data gets bigger, the curve gets flatter.



Linear growth rate

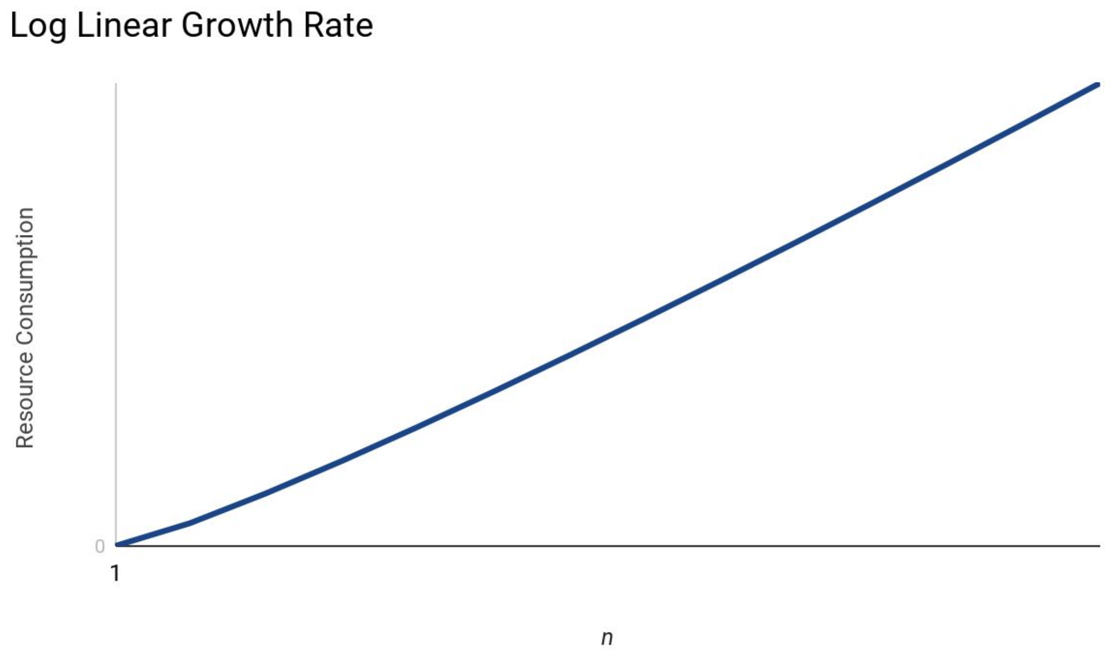
The resource needs and the amount of data is directly proportional to each other.



Log linear growth rate

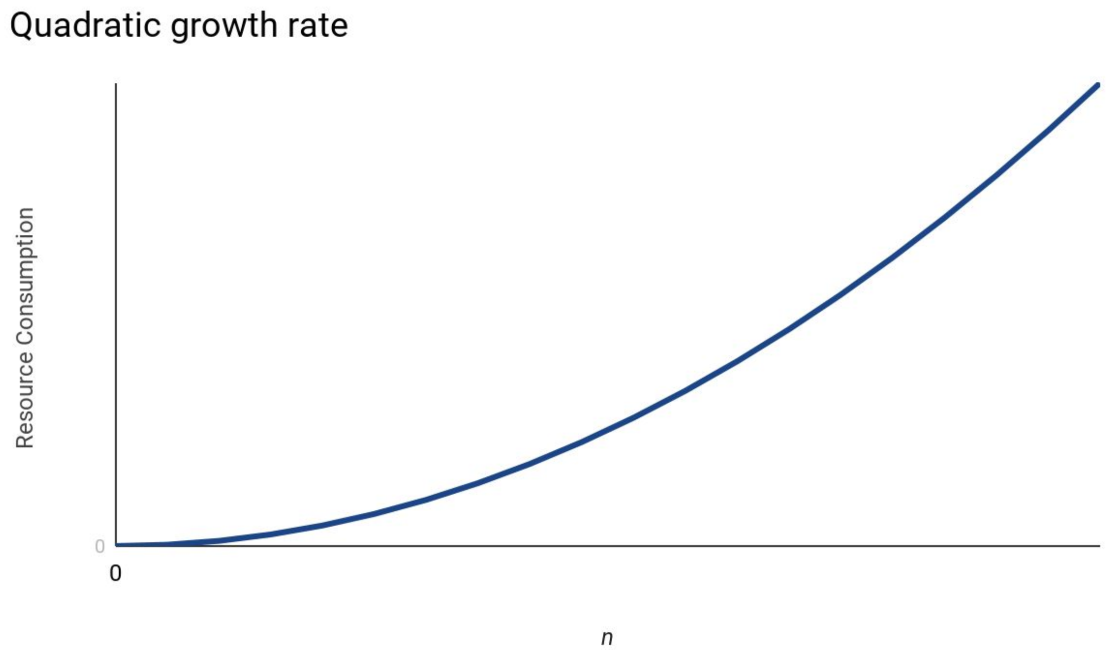
Slightly curved line.

The curve is more pronounced for lower values than higher.



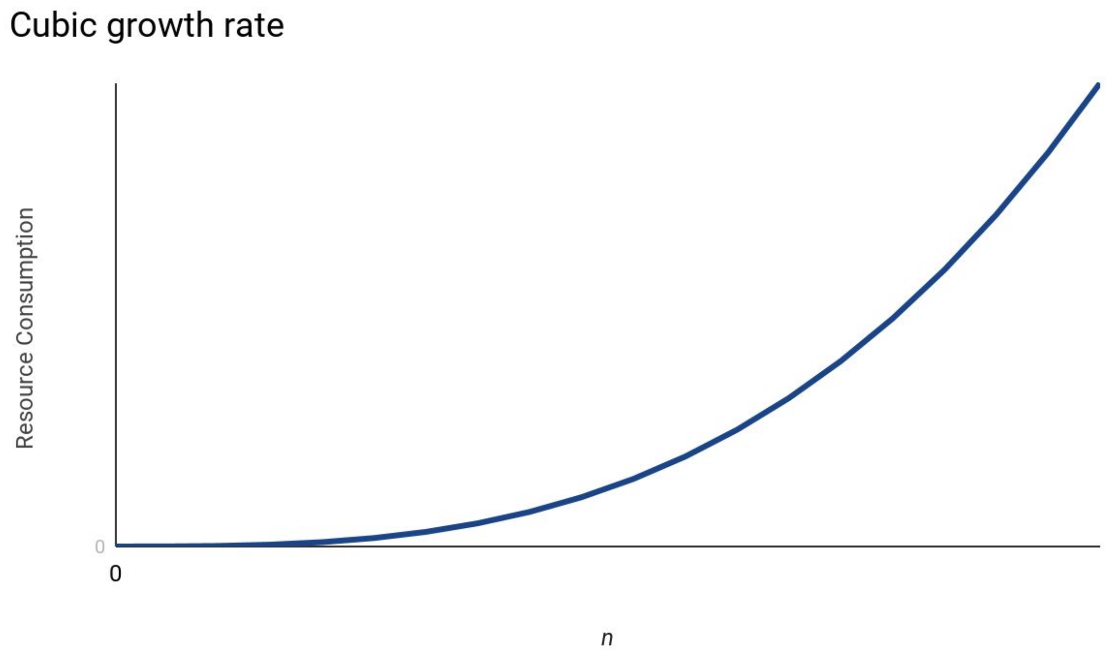
Quadratic growth rate

The resource needs is proportional to the square of the amount of data.



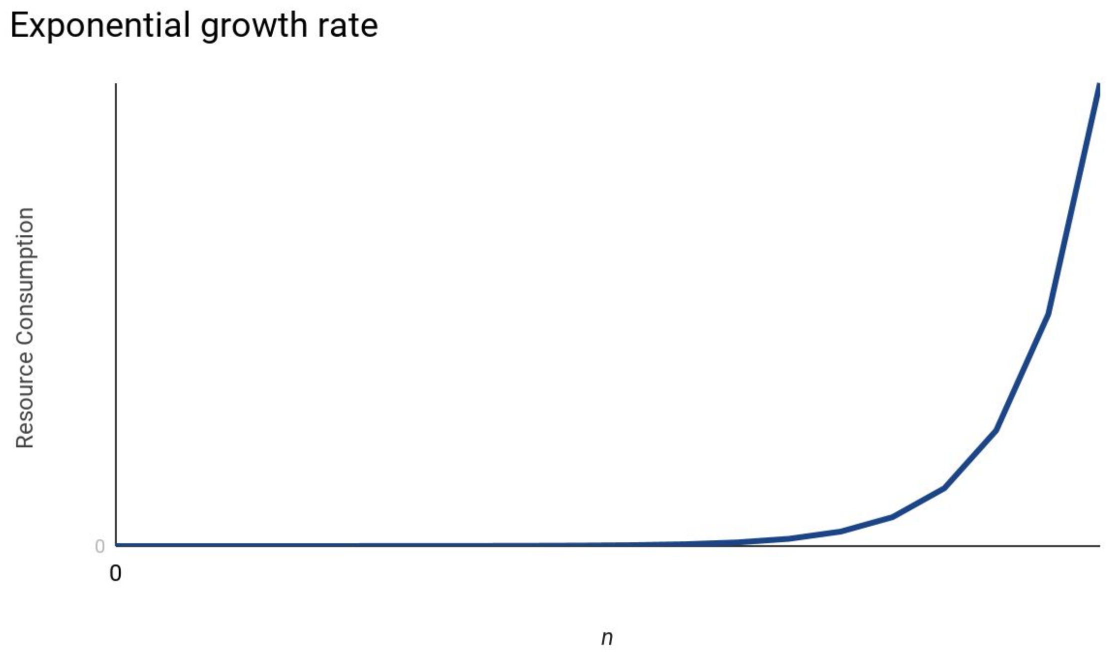
Cubic growth rate

Similar to quadratic but grows significantly faster.



Exponential growth rate

Each extra unit of data requires a doubling of resources. (2^n)



Asymptotic analysis

"How does the running time scale with the size of the input?"

The idea is that we ignore low-order terms and constant factors, focusing instead on the shape of the running time curve.

Typical notation:

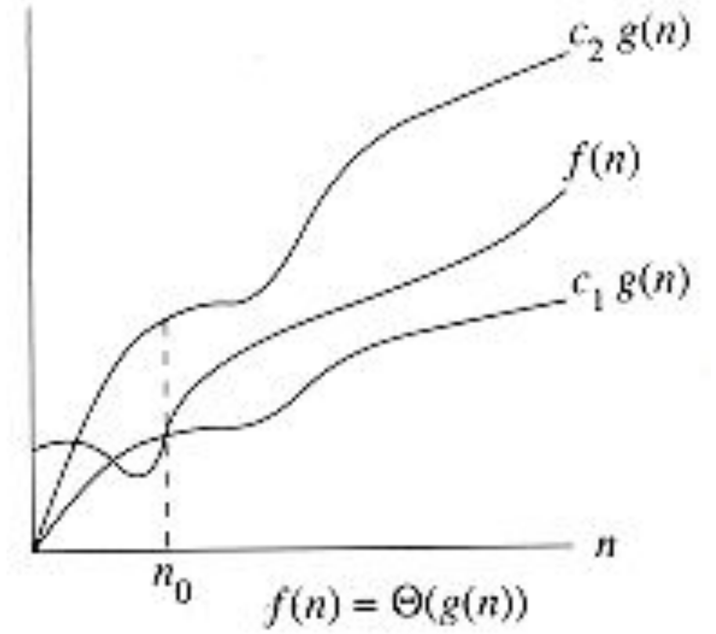
- n = The size of the input
- T(n) = The running time of our algorithm on an input of size n.

Θ-notation

For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that
$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$

Informally, " T(n) is Θ(f(n)) " basically means that the function f(n) describes the exact (**asymptotically tight**) bound for T(n).

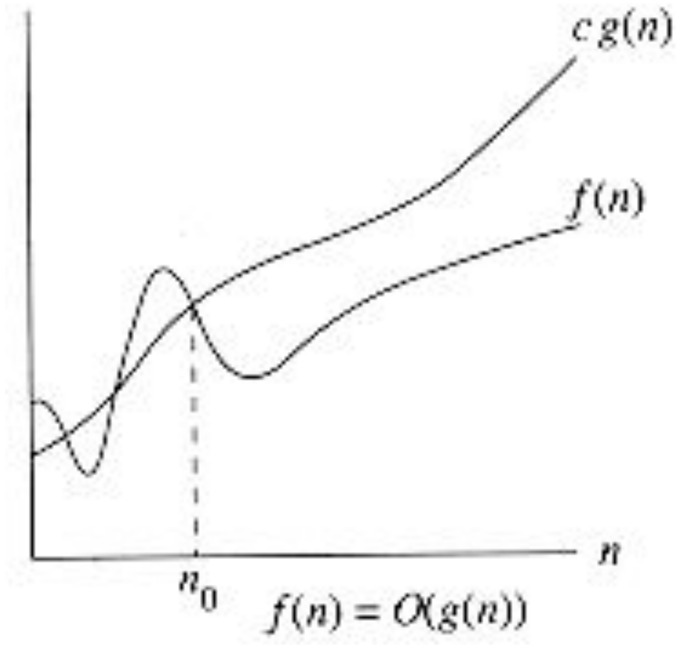


O-notation

Provides an **asymptotic upper bound** on a function.

For a given function g(n), we denote by O(g(n)) the set of functions

O(g(n)) = {f(n) : there exist positive constants c and n_0 such that $0 \leq f(n) \leq c.g(n)$ for all $n \geq n_0$}.

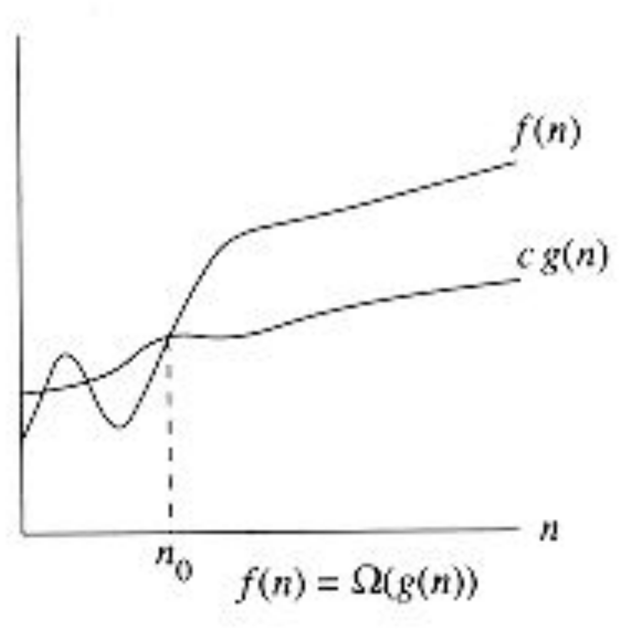


Ω-notation

Provides an **asymptotic lower bound** on a function.

For a given function g(n), we denote by Ω(g(n)) the set of functions

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq c.g(n) \leq f(n)$ for all $n \geq n_0\}$.



o-notation and ω-notation

o-notation provides an upper bound that is **not** asymptotically tight.

$o(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) < c.g(n)$ for all $n \geq n_0\}$.

ω-notation provides a lower bound that is **not** asymptotically tight.

$\omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq c.g(n) < f(n)$ for all $n \geq n_0\}$.

Readings

Chapter 1 from Horowitz et al.