Faculty Development Program

Design and Analysis of Algorithms

Asymptotic Analysis of Algorithms

R. S. Milton

Department of Computer Science and Engineering

SSN College of Engineering

11 December, 2014

Contents

Thirukkural 479

The prosperity of him who lives without knowing the measure (of his wealth), will perish, even while it seems to continue.

Design and analysis of algorithm

- Specify the problem precisely.
- Design an algorithm for the problem.
- Construct the algorithm correctly.
- Analyze running time of the algorithm.
    - Iterative algorithms
    - Recursive algorithms
    - Amortized analysis

1. Analysis of algorithms

- Execution of an algorithm requires resources such as time, memory, and disk space.
- All resources are limited.
- The amounts of resources the algorithm required.
- Performance (efficiency, complexity) of algorithm is measured by the resources the algorithm requires for its execution.

Analysis

Calculating analytically the resources such as time and memory required for executing an algorithm without actually executing the algorithm is called analysis of algorithm.

Measures of performance

- Space complexity: memory space required to execute the algorithm.
- Time complexity: Running time required to execute the algorithm.

Primarily interested in the running time of algorithms:

- Running time is the most important of the two resources.
- Running time is more tractable analytically.

Use of analysis

- Efficiency: The running time of the algorithm should be practical (polynomial) — it depends on the size of the input, the nature of the input, and the machine which executes the algorithm.
- Comparison: When we have alternative algorithms to solve the same problem, we want to compare their performances (resource requirements), and choose the one that best suits the available resources.

2. Model of machine

- An algorithm is composed of instructions which can be executed by a machine.
- The time required to execute an instruction depends on the machine — its instruction set, speed.
- Choose a machine.
- Analysis becomes intractable for actual machines.
- Running time thus obtained has more details than is necessary. Therefore, it suffices to consider a simplified abstract model of an actual machine.

Actual machine

- Analysis tedious
- Too many details

2.1. RAM model of computation

- Defines a set of basic instructions or basic steps.
- Instructions of all the algorithms can be decomposed to the basic steps of this abstract machine.

Basic step and analysis

- Executed in constant amount of time. It does not depend on the input size.
- Analyzing an algorithm $\equiv$ counting the total number of basic steps executed by the algorithm.

2.2. RAM model of computation

Random Access Machine

- Basic instructions or basic steps — take constant time, independent of input size.
  - Arithmetic
  - Comparison
  - Logical
  - Assignment
- Loops and subroutine (function) calls take varying time
- One level memory hierarchy — instructions and data are in memory.

3. Running time

Running time of an algorithm $\equiv$ the total number of basic steps (basic instructions) executed.

We have abstracted away the architecture, instruction set and the speed of the computer.

- Calculate how many times each basic step is executed.
- Running time $T(n)$ is the total number of basic steps executed by the algorithm.
- $T(n)$ depends on the size of the input $n =$ input size or problem size.
- For the same problem size, the running time also depends on the properties of the input (e.g. is it ordered?)

Analysis of algorithm – Summary

- Calculate the
  - Memory space
  - Running time

  required to execute the algorithm to completion, without executing the algorithm.
- Why analyze algorithms?
  - Efficiency
  - Comparison
- How to analyze an algorithm?
  - Identify the basic statements.
  - Count how many times each basic statement is executed.
  - Total the count.

4. Analysis of running time — illustration

Algorithm: Sum(a)

Input: Array $a[0:n]$ of n numbers

Output: s, the sum of the numbers of $a[0:n]$

```
1 $s \leftarrow 0$                          // $c_1$
2 $i \leftarrow 0$                          // $c_2$
  -- s = sum []
3 until $i = n$ do                          // $c_3 \times (n+1)$
4   | $s \leftarrow s + a[i]$               // $c_4 \times n$
5   | $i \leftarrow i + 1$                  // $c_5 \times n$
6 end
  -- s = sum [0:n]
7 return s
```

Running time of Sum is

$$\begin{aligned} T(n) &= c_1 + c_2 + c_3(n+1) + c_4 n + c_5 n \\ &= (c_3 + c_4 + c_5)n + (c_1 + c_2 + c_3) \\ &= an + b \end{aligned}$$

where a, b are the constants

$$\begin{aligned} a &= c_3 + c_4 + c_5 \\ b &= c_1 + c_2 + c_3 \end{aligned}$$

Algorithm: LinearSearch (a, x)

Input: Array $a[0:n]$ of n numbers, and a target x to search for.

Output: i, such that $a[i] = x$ if x is in the array; otherwise, $i = n$

```
1 $i \leftarrow 0$                                  // $c_1$
  -- x $\notin$ []
2 until $i = n$ or $a[i] = x$ do                    // $c_2 \times (n+1)$
3 |  $i \leftarrow i + 1$                           // $c_3 \times n$
4 end
  -- x $\notin$ [0:i]
5 return i
```

Worst case occurs when the target is not found. Worst-case running time of LinearSearch is

$$\begin{aligned} T(n) &= c_1 + c_2(n+1) + c_3 n \\ &= (c_2 + c_3)n + c_1 \\ &= an + b \end{aligned}$$

where a, b are the constants

$$\begin{aligned} a &= c_2 + c_3 \\ b &= c_1 \end{aligned}$$

Best case occurs when the target is the 0th item. Best-case running time of LinearSearch is

$$\begin{aligned} T(n) &= c_1 + c_2 \\ &= a \end{aligned}$$

Average case is difficult to define. The target is equally likely to be in any of the n positions: 0, 1, ..., $n-1$.

$$P(0) = P(1) = \ldots = P(n-1) = \frac{1}{n}$$

On an average, the number of iterations

$$\begin{aligned}
&= P(0) \times 0 + P(1) \times 1 + P(2) \times 2 + \ldots + P(n-1) \times (n-1) \\
&= \frac{1}{n} \times (1 + 2 + \ldots + (n-1)) \\
&= \frac{1}{n} \times \sum_{i=1}^{n-1} i \\
&= \frac{1}{n} \times \frac{(n-1)n}{2} \\
&= \frac{n-1}{2}
\end{aligned}$$

On an average, the number of times the loop condition is tested

$$= \frac{n-1}{2} + 1 = \frac{n+1}{2}$$

Algorithm: LinearSearch (a, x)

Input: Array $a[0:n]$ of n numbers, and a key x to search for.

Output: i, such that $a[i] = x$ if x is in the array; otherwise, $i = n$

```
1 $i \leftarrow 0$                                  // $c_1$
2 until $i = n$ or $a[i] = x$ do                    // $c_2 \times \frac{n+1}{2}$
3     $i \leftarrow i + 1$                          // $c_3 \times \frac{n-1}{2}$
4 end
5 return i
```

Average-case running time of LinearSearch is

$$\begin{aligned} T(n) &= c_1 + c_2\left(\frac{n+1}{2}\right) + c_3\left(\frac{n-1}{2}\right) \\ &= (c_2+c_3)n + \left(c_1 - \frac{c_2}{2} - \frac{c_3}{2}\right) \\ &= an + b \end{aligned}$$

where a, b are the constants $a = c_2 + c_3,\ b = c_1 - \frac{c_2}{2} - \frac{c_3}{2}$.

Worst-case	Best-case	Average-case
$an + b$	a	$an + b$
$O(n)$	$O(1)$	$O(n)$
Linear time	Constant time	Linear time

5. Running time of Selection sort

Algorithm: SelectionSort(a)

Input: An array $[a_0, a_1, \dots a_{n-1}]$ of size n

Output: A permuted array $[a'_0, a'_1, \dots, a'_{n-1}]$ such that $a'_0 \leq a'_1 \leq \dots \leq a'_{n-1}$

```
1  for $i \leftarrow 1$ to $n-1$ do                 // $c_1 \times n$
2      $m, j \leftarrow i, i+1$                      // $c_2 \times (n-1)$
3      until $j = n$ do                            // $c_3 \times \sum_{i=1}^{n-1} n - i$
4          if $a[j] < a[m]$ then                   // $c_4 \times \sum_{i=1}^{n-1} n - i - 1$
5              $m \leftarrow j$                      // $c_5 \times \sum_{i=1}^{n-1} n - i - 1$
6          end
7          $j \leftarrow j + 1$                      // $c_6 \times \sum_{i=1}^{n-1} n - i - 1$
8      end
9      swap $a[i]$ $a[m]$                          // $c_7 \times (n-1)$
10 end
```

Running time analysis of Insertion sort

- The outer loop iterates $n-1$ times.
- For each iteration of the outer loop, the nested loop iterates i times.
- Therefore, the nested loop body is executed $\Sigma_{i=1}^{n-1} i$ times.

$$\begin{aligned}\sum_{i=1}^{n-1} i &= 1+2+3+\ldots+(n-1) \\ &= \frac{(n-1)\times n}{2} \\ &= \frac{n^2}{2}-\frac{n}{2}\end{aligned}$$

$$\begin{aligned}\sum_{i=1}^{n-1} (i+1) &= 2+3+\ldots+n \\ &= \frac{n\times(n+1)}{2}-1 \\ &= \frac{n^2}{2}+\frac{n}{2}-1\end{aligned}$$

The running time of the algorithm is

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1}(i+1) + c_5 \sum_{i=1}^{n-1} i \\
&\quad + c_6 \sum_{i=1}^{n-1} i + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n^2}{2} + \frac{n}{2} - 1) + \\
&\quad (c_5 + c_6)(\frac{n^2}{2} - \frac{n}{2}) + c_7(n-1) \\
&= (\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2})n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n - \\
&\quad (c_2 + c_3 + c_4 + c_7) \\
&= an^2 + bn + c
\end{aligned}
$$

where a, b, c are the constants

$$
\begin{aligned}
a &= \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} \\
b &= c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} + c_7 \\
c &= c_2 + c_3 + c_4 + c_7
\end{aligned}
$$

6. Best-case and average-case running time

Worst-case running time

When the input array is reverse-sorted

- key to be inserted in an appropriate position in the sorted subarray $A[0:i]$ is less than all the items in the sorted subarray.
- Therefore, key will have to be inserted as the 0th item in the subarray $A[0:i]$
- Hence, for each outer loop iteration, the nested loop is iterated i times
- Totally, the nested loop is iterated $\sum_{i=1}^{n} i$ times (and the nested loop condition, $\sum_{i=1}^{n} i + 1$ times).

Worst-case running time

Worst-case running time is the longest running time on any input of size n

- It provides a guarantee on the upper-bound of the running time.
- It occurs often.
- The average-case is as bad as worst-case.
- It is mathematically more tractable than average case.

Best-case running time

Best-case of the Insertion sort occurs when the input array is already sorted,

- key is already in its right position.
- Hence, the nested loop terminates immediately (iterated 0 times)
- Totally, the nested loop is executed $\sum_{i=1}^{n} 1 = n$ times.

Input: An array $[a_0, a_1, \ldots, a_{n-1}]$ of size n
Output: A permuted array $[a_0^{'}, a_1^{'}, \ldots, a_{n-1}^{'}]$ such that $a_0^{'} \leq a_1^{'} \leq \ldots \leq a_{n-1}^{'}$

```
for $i \leftarrow 1$ to $n-1$ do                     // $c_1 \times n$
    $j \leftarrow i$                                   // $c_3 \times (n-1)$
    until $j = 0$ or $a[j-1] \leq a[i]$ do             // $c_4 \times \sum_{i=1}^{n-1} 1$
        $a[j] \leftarrow a[j-1]$                       // $c_5 \times 0$
        $j \leftarrow j-1$                             // $c_6 \times 0$
    end
    $a[j] \leftarrow a[i]$
end
```

In insertion sort, when the input array is already sorted, the key to be inserted in the sorted section of the array is already in its right position, and hence, right after one probe, the nested loop terminates.

If the outer loop iterates n times, then the nested loop statement is executed n times; in each of the n times, the nested loop condition is evaluated only once and the loop terminates immediately. Therefore, in all, the nested loop condition is evaluated

$$\sum_{i=1}^{n} 1 = n$$

times.

The best-case running time of the algorithm is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \\ &= bn + c \end{aligned}$$

Thus, the order of growth of the best-case running time of insertion sort is $\Theta(n)$.

Average-case running time

- On an "average", in the sorted section $a[0:i]$, half the items are greater and half the items are less than $a[i]$
- What is "average" input?
- All inputs of a given size n equally likely!

Worst-case	Best-case	Average-case
$an^2 + bn + c$	$an + b$	$an^2 + bn + c$
$O(n^2)$	$O(n)$	$O(n^2)$
Quadratic time	Linear time	Quadratic time

Exercises

1. What are the **goals of algorithm analysis**?
2. State two resources an algorithm requires for execution.
3. What are the two measures of performance of algorithms?
4. What is analysis of algorithm?
5. When is an algorithm said to have a practical running time?
6. What is meant by polynomial time?
7. How is a basic step of the abstract machine is defined?
8. How is an algorithm analyzed in the abstract machine?
9. How is the running time of algorithm calculated in RAM machine?
10. State the two factors the running time of an algorithm depends on.
11. Analyze the running time of insertion sort.
12. Why is analyzing the worst-case running time preferred to other cases?
13. Calculate average-case running time of Insertion sort.
14. How do you calculate the order of growth of a function?

15. Mention the abstractions we have made to calculate the running time of algorithms.
16. Define order of growth or asymptotic growth of a function.

Calculate the number of basic steps executed in each of the following algorithms:

1. How many times the `for` loop body is executed?

```
$sum \leftarrow 0$
for $i \leftarrow 0$ to $n-1$
  $sum \leftarrow sum + a[i]$
end
```

2. How many times the `for` loop body is executed?

```
$i \leftarrow n$
until $i = 0$ or $a[i-1] \leq a[i]$ do
  swap $a[i-1], a[i]$
  $i \leftarrow i - 1$
end
```

3. How many times the nested `for` loop body is executed?

```
for $i \leftarrow 1$ to $n-1$
    $j \leftarrow i$
    until $j = 0$ or $a[j] > a[i]$ do
        swap $a[j-1]$ $a[j]$
        $j \leftarrow j - 1$
    end
end
```

4. How many times the `for` loop body is executed?

```
$sum \leftarrow 0$
for $i \leftarrow 0$ to $n-1$
    $sum \leftarrow sum + a[i]$
end
```

5. How many times the statement $min \leftarrow i$ is executed?

```
$min \leftarrow 0$
for $i \leftarrow 0$ to $n-1$
    if $a[i] < a[min]$ then
        $min \leftarrow i$
    end
end
```

6. Calculate how many times the innermost assignment is repeated?

```
for $i \leftarrow 1$ to $m$
    for $j \leftarrow 1$ to $n$
        $c[i,j] \leftarrow a[i,j] + b[i,j]$
    end
end
```

7. How many times the innermost statement (increment x) is executed? What is the final value of x? Express the time complexity $T(n)$ in big Oh.

```
$x \leftarrow 0$
for $i \leftarrow 1$ to $n$
    for $j \leftarrow i$ to $2n$
        $x \leftarrow x + 1$
    end
end
```

8. How many times the innermost loop is iterated?

```
for $i \leftarrow 1$ to $m$
    for $j \leftarrow 1$ to $p$
        $c[i,j] \leftarrow 0$
        for $k \leftarrow 1$ to $n$
            $c[i,j] \leftarrow c[i,j] + a[i,k] * b[k,j]$
        end
    end
end
```

Analysis of running time

- Construct algorithm = design, prove the correctness, analyze algorithm.
- Analyze algorithm = calculate the running time analytically, without executing it.
- Running time depends on the input size, nature of the input, and the machine executing the instructions.
- Simplified model (abstraction) of machine — RAM: basic instructions execute in constant amount of time, independent of the input (operand) size.
- Calculate the total number of times each basic step in the algorithm is executed.
- If a basic step is inside a loop, calculate how many times the loop is iterated.
- Analyze the worst-case running time.

7. Order of growth, informally

Big picture

Our life is frittered away by detail. Simplify, simplify. (Henry David Thoreau)
Do not miss the wood for the trees.(Proverb)

In order to calculate the order of growth of the function $an^2 + bn + c$,

- ignore lower-order terms such as bn, c
- ignore multiplicative constants such as a of the highest order term an^2

The running time $T(n) = (an^2 + bn + c)$ of the algorithm $= \Theta(n^2)$

- For small inputs, this running time will have errors
- For large inputs, this running time is valid for both efficiency and comparison purposes.

7.1. Order of growth, informally

- Running time of algorithms — how do they grow when the input size becomes very large?
- We are interested in the asymptotic growth of functions, growth of functions in the limit.
- A simple way to calculate the asymptotic growth of a function is to abstract away its low-order terms and constant factors of the highest order term.
- Running times of algorithms are expressed by their asymptotic growth.
- Compare "growth" of functions:

O	$\leq$
Ω	$\geq$
Θ	$=$
o	$<$
ω	$>$

We use asymptotic notation to compare "growths" of functions:

Asymptotic notation	Intuitive meaning	Order of growth
$f = O(g)$	$f \leq g$	f grows slower than or as fast as g
		f grows no faster than g
$f = \Omega(g)$	$f \geq g$	f grows faster than or as fast as g
		f grows no slower than g
$f = \Theta(\mathrm{g})$	$f = g$	f grows as fast as g
		f grows at the same rate as g
$f = o(g)$	$f < g$	f grows slower than g
$f = \omega(g)$	$f > g$	f grows faster than g

8. Asymptotic growth, or order of growth

- As input size n increases, the running time $T(n)$ also increases.
- Measure of how fast it increases with the input size n. The running times $T(n) = n^2$ and $T(n) = n^3$ both increase with the input size. But how fast?
- Asymptotic growth (order of growth) of a function $f(n)$ can be observed only when n becomes very large.

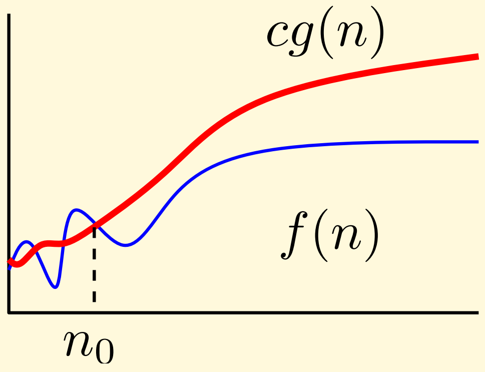
Asymptotic growth or Order of growth

- As input size n increases, the running time $T(n)$ also increases.
- How fast the running time $T(n)$ grows as n becomes very large, tends to infinity, in the limit.

9. Calculation of asymptotic growth

- Running time is a function of input size n, $T(n)$.
- Ignore lower-order terms.
- Ignore multiplicative constants of the highest order term
- Find out a suitable constant.

Big-Oh notation



- Growth of a function $f(n)$ can be described by an asymptotic upper bound $g(n)$ for it.
- Informally, we say that the given function is “less than or equal to” the asymptotic upper bound, or, intuitively, the given function grows slower than or as fast as the asymptotic upper bound; $f(n) \leq g(n)$. Formally, $f(n) = O(g(n)$.
- Precisely, there exists a constant c such that $f(n) \leq cg(n)$.
- Even this needs to be true only when n is very large, that is, for all $n \geq n_0$.

Big-Oh

Working definition

$$f(n) = O(g(n)) \equiv \text{There exist constants } c \text{ and } n_0 \text{ such that } f(n) \leq cg(n) \text{ for all } n \geq n_0$$

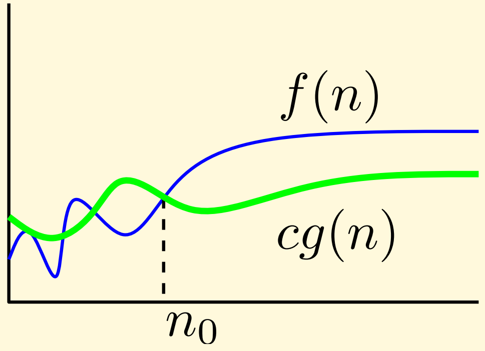
Or, more precisely,

$$O(g(n)) = \{f(n) : \text{There exist +ve constants } c \text{ and } n_0 \text{ such that } f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

Therefore,

$$f(n) = O(g(n)) \equiv f(n) \in O(g(n))$$

Big-Omega notation



- Growth of a function $f(n)$ can be described by an asymptotic lower bound $g(n)$ for it.
- Intuitively, we say that the given function is "greater than or equal to" the asymptotic lower bound, or the given function grows faster than or as fast as the asymptotic lower bound; $f(n) \geq g(n)$. Formally, $f(n) = \Omega(g(n))$.
- Precisely, there exists a constant c such that $f(n) \geq cg(n)$.
- Even this needs to be true only when n is very large, that is, for all $n \geq n_0$.

Big-Omega

$$f(n) = \Omega(g(n)) \equiv \quad \text{There exist constants } c \text{ and } n_0 \text{ such that} \\ f(n) \geq cg(n) \text{ for all } n \geq n_0$$

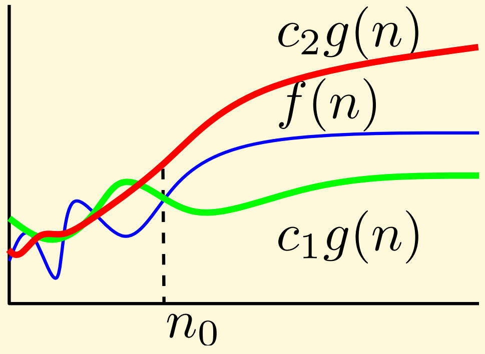
Precisely,

$$\Omega(g(n)) = \{f(n) : \text{There exist +ve constants } c \text{ and } n_0 \\ \text{such that } f(n) \geq cg(n) \text{ for all } n \geq n_0\}$$

Therefore,

$$f(n) = \Omega(g(n)) \equiv f(n) \in \Omega(g(n))$$

Big-Theta notation



- Growth of a function $f(n)$ can be described by an asymptotic tight bound $g(n)$ for it.
- Intuitively, we say that the given function is "equal to" the asymptotic tight bound, or the given function grows as fast as the asymptotic tight bound: $f(n) = g(n)$; formally, $f(n) = \Theta(g(n))$.
- Precisely, there exists constants c_1 and c_2 such that $c_1g(n) \leq f(n) \leq c_2g(n)$.
- Even this needs to be true only when n is very large, that is, for all $n \geq n_0$.

Big-Theta notation

$$f(n) = \Theta(g(n)) \equiv \quad \text{There exist constants } c_1, c_2, n_0 \text{ such that}$$
$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0$$

Precisely,

$$\Theta(g(n)) = \{f(n) : \text{There exist +ve constants } c_1, c_2 \text{ and } n_0$$
$$\text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$
$$\text{for all } n \geq n_0\}$$

Therefore,

$$f(n) = \Theta(g(n)) \equiv f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

o notation

$$\begin{aligned} o(g(n)) = \{f(n) :\ & \text{for all constants } c > 0, \text{ there exists a} \\ & \text{constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \\ & \text{for all } n \geq n_0\} \end{aligned}$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

ω notation

$$\begin{aligned} \omega(g(n)) = \{f(n) :\ & \text{for all constants } c > 0, \text{ there exists a} \\ & \text{constant } n_0 > 0 \text{ such that } 0 \leq f(n) > cg(n) \\ & \text{for all } n \geq n_0\} \end{aligned}$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Asymptotic notation in equations

- On right side: $O(g(n))$ stand for some anonymous function in $O(g(n))$.
- On left side: No matter how the anonymous function is chosen on the left side, there is a way to choose an anonymous function on the right side to make the equation valid.

10. Common asymptotic growth functions

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 yrs
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15}
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4 × 1013 yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Constant functions	$g(n) = 1$
Logarithmic functions	$g(n) = \lg n$
Linear functions	$g(n) = n$
Superlinear functions	$g(n) = n \lg n$
Quadratic functions	$g(n) = n^2$
Cubic functions	$g(n) = n^3$
Exponential functions	$g(n) = c^n$
Factorial functions	$g(n) = n!$

Dominance relations and efficiency

$$1 \leq \lg n \leq n \leq n \lg n \leq n^2 \leq n^3 \leq c^n \leq n!$$

If the asymptotic growth of a running time is “less than equal” to polynomial time (n^m), then the running time is practically efficient. If the asymptotic growth of a running time is “greater than” polynomial time, it is not practical.

11. Exercises

- Which of the functions is greater, and in what range of n? $f_1(n) = n^2, f_2(n) = 2n + 20$. Find out a constant c so that $f_2 \leq cf_1$.
- Which of the functions is greater, and in what range of n? $f_3(n) = n + 1, f_2(n) = 2n + 20$. Find out a constant c so that $f_3 \leq cf_2$.
- $3n^2 - 100n + 6 = O(n^2)$. Choose c.
- $3n^2 - 100n + 6 = O(n^3)$. Choose c and n_0.
- $3n^2 - 100n + 6 \neq O(n)$.
- $3n^2 - 100n + 6 = \Omega(n^2)$. Choose c and n.
- $3n^2 - 100n + 6 \neq \Omega(n^3)$. Choose c and n.
- $3n^2 - 100n + 6 = \Omega(n)$. Choose c.
- $3n^2 - 100n + 6 = \Theta(n^2)$.

- $3n^2 - 100n + 6 \neq \Theta(n^3)$.
- $3n^2 - 100n + 6 \neq \Theta(n)$.
- $2^{n+1} = \Theta(2^n)$.

12. Properties of asymptotic functions

Comparison of functions — Relational properties:

- Transitivity
  $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$.
  Same for O, Ω, o, ω
- Reflexivity
  $f(n) = \Theta(f(n))$. Same for O, Ω
- Symmetry:
  $f(n) = \Theta(g(n)) \equiv g(n) = \Theta(f(n))$.
- Transpose symmetry
  $f(n) = O(g(n)) \equiv g(n) = \Omega(f(n))$.
  $f(n) = o(g(n)) \equiv g(n) = \omega(f(n))$.
- Comparisons
  $f(n)$ is asymptotically smaller than $g(n)$ if $f(n) = o(g(n))$.
  $f(n)$ is asymptotically larger than $g(n)$ if $f(n) = \omega(g(n))$.

Adding two functions

$$O(f(n)) + O(g(n)) \equiv O(\max(f(n), g(n)))$$
$$\Omega(f(n)) + \Omega(g(n)) \equiv \Omega(\max(f(n), g(n)))$$
$$\Theta(f(n)) + \Theta(g(n)) \equiv \Theta(\max(f(n), g(n)))$$

Multiplying functions

$$O(cf(n)) \equiv O(f(n))$$
$$\Omega(cf(n)) \equiv \Omega(f(n))$$
$$\Theta(cf(n)) \equiv \Theta(f(n))$$
$$O(f(n)) \times O(g(n)) \equiv O(f(n) \times g(n))$$
$$\Omega(f(n)) \times \Omega(g(n)) \equiv \Omega(f(n) \times g(n))$$
$$\Theta(f(n)) \times \Theta(g(n)) \equiv \Theta(f(n) \times g(n))$$