Algorithm Efficiency

Big-O Notation

What is the algorithm's efficiency

- The algorithm's efficiency is a function of the number of elements to be processed. The general format is

$$f(n) = \text{efficiency}$$

The basic concept

- When comparing two different algorithms that solve the same problem, we often find that one algorithm is an order of magnitude more efficient than the other.
- If the efficiency function is linear,
  - This means that the algorithm is linear and it contains no loops or recursions.
  - In this case, the algorithm's efficiency depends only on the speed of the computer.

The basic concept

- If the algorithm contains loops or recursions (any recursion may always be converted to a loop),
  - It is called nonlinear.
  - In this case, the efficiency function strongly and informally depends on the number of elements to be processed.

Linear Loops

- The efficiency depends on how many times the body of the loop is repeated. In a linear loop, the loop update (the controlling variable) either adds or subtracts.
  - For example:

```
for (i ← 0 step 1 to 1000)
        the loop body
```

    - Here the loop body is repeated 1000 times
    - The efficiency is directly proportional to the number of iteration, it is: $f(n) = n$

Logarithmic Loops

- In a logarithmic loop, the controlling variable is multiplied or divided in each iteration
  - For example:

```
Multiply loop
  for (i ← 1 step × 2 to 1024)
        the loop body

Divide loop
  for (i ← 1024 step /2 down to 1)
        the loop body
```

- For the logarithmic loop the efficiency is determined by the following formula: $f(n) = \log n$

Logarithmic Loops

- Analysis of multiply and divide loops

Multiply		Divide	
iteration	Value of `i`	Iteration	Value of `i`
1	1	1	1024
2	2	2	512
3	4	3	256
4	8	4	128
5	16	5	64
6	32	6	32
7	64	7	16
8	128	8	8
9	256	9	4
10	512	10	2
(exit)	1024	(exit)	1

Linear Logarithmic Nested Loop

- A total number of iterations in the linear logarithmic nested loop is equal to the product of the numbers of iterations for the external and inner loops, respectively
  - For example:

```
for (i ← 1 to 10)
   for (j ← 1 step × 2 to 10)
             the loop body
```

    - The outer loop updates either adds or subtracts, while the inner loop multiplies or divides ($10 \times \log 10$ in our example)
    - For the linear logarithmic nested loop the efficiency is determined by the following formula: $f(n) = n \log n$

Quadratic Nested Loop

- A total number of iterations in the quadratic nested loop is equal to the product of the numbers of iterations for the external and inner loops, respectively
  - For example:

```
for (i ← 1 to 10)
   for (j ← 1 to 10)
          the loop body
```

    - Both loops in this example add ($10 \times 10 = 100$ in our example)
    - For the quadratic nested loop the efficiency is determined by the following formula: $f(n) = n^2$

Dependent Quadratic Nested Loop

- A total number of iterations in the dependent quadratic nested loop is equal to the product of the numbers of iterations for the external and inner loops
  - For example:

```
for (i ← 1 to 10)
   for (j ← i to 10)
          the loop body
```

    - The number of iterations of the inner loop depends on the outer loop. It is equal to the sum of the first n members of an arithmetic progression: $n(n+1)/2$
    - For the dependent quadratic nested loop the efficiency is determined by the following formula: $\boldsymbol{f(n) = n(n+1)/2}$

Big-O notation

- The number of statements executed in the function for ***n*** elements of data is a function of the number of elements expressed as ***f*(*n*)**.
- Although the equation derived for a function may be complex, a ***dominant factor*** in the equation usually determines ***the order of magnitude*** of the result.
- This factor is a **big-O**, as in "on the order of". It is expressed as **O(*n*)** .

Big-O notation

- The big-O notation can be derived from $f(n)$ using the following steps:
  - In each term set the coefficient of the term to 1.
  - Keep the largest term in the function and discard the others. Terms are ranked from lowest to highest:
    $\log n, n, n \log n, n^2, n^3, \ldots, n^k, \ldots, 2^n, \ldots, n!$

$$f(n) = n\left(\frac{n+1}{2}\right) = \frac{1}{2}n^2 + \frac{1}{2}n \quad \Rightarrow \quad n^2 + n \quad \Rightarrow \quad f(n) = O(n^2)$$

Measures of Efficiency

- n = 10,000

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	Microseconds
Linear	$O(n)$	10,000	Seconds
Linear logarithmic	$O(n(\log n))$	140,000	Seconds
Quadratic	$O(n^2)$	$10{,}000^2$	Minutes
Polynomial	$O(n^k)$	$10{,}000^k$	Hours
Exponential	$O(c^n)$	$2^{10{,}000}$	Intractable
Factorial	$O(n!)$	10,000!	Intractable

Algorithm Efficiency

Big-O Notation and
Other Bound Notations

Insertion Sort 2nd algorithm

```
for j ← 1 to length[A]–1
  do
    { key ← A[j]
      for i ← j + 1 to length[A]
        do
        { if A[j] > A[i]
            then   key ← A[j]
                   A[j] ← A[i]
        }
        A[i] ← key
    }
```

Insertion Sort 2nd algorithm: the worst case

for j ← 1 to **length**[A]–1	$c_1 n$
do	
{ **key** ← $A[j]$	$c_2(n-1) \approx c_2 n$
for i ← j + 1 to **length**[A]	$\sim c_3(n^2/2)$
do	
{ **if** $A[j] > A[i]$	$\sim c_4(n^2/2)$
then key ← $A[j]$	$\sim c_5(n^2/2)$
$A[j]$ ← $A[i]$	$\sim c_6(n^2/2)$
}	
$A[i]$ ← key	$\sim c_7(n^2/2)$
}	

Insertion Sort 2nd algorithm: the worst case

$$T(n) = c_1 n + c_2 n + (c_3 + c_4 + c_5 + c_6 + c_7)\frac{n^2}{2} =$$

$$= \left(\frac{c_3}{2} + \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + (c_1 + c_2)n$$

- Thus in the terms of Big-O notation:

$$T(n) = an^2 + bn \approx n^2 + n \quad \Rightarrow \quad \boxed{T(n) = O\left(n^2\right)}$$

Growth of Functions: Asymptotic Bound Notations

- That algorithm is more efficient whose efficiency (as a function of the input size) growths slowly.

- Thus, to compare the efficiency of two or more different algorithms it is enough to compare their efficiencies in terms of growth of functions

Growth of Functions: Asymptotic Bound Notations

- We say that in terms of big-O notation the sorting running time is **$O(n^2)$** for any of those 3 sorting algorithms, which we considered

- How we can estimate the running time function in terms of growth of functions depending on the input size without direct estimation of the cost of each step of an algorithm (the cost of statement in the pseudocode description of an algorithm or a block in the flow chart)?

Growth of Functions: Asymptotic Bound Notations

- Let ***f*(*n*)** be a running time function (the efficiency function) whose equation is *unknown* or is *difficult to evaluate* (a long equation containing a number of additive terms, which in turn contain additive and multiplicative terms where a problem size appears raised in different degrees).

Growth of Functions: Asymptotic Bound Notations

- If it is possible to prove that ***f*(*n*)** behaves similarly to some *well known* function ***g*(*n*)**
  - For example
    - $f(n)$ growths not faster than $g(n)$
    - $f(n)$ growths not slower than $g(n)$
    - $f(n)$ growths as quickly as $g(n)$
- then we can evaluate behavior of ***f*(*n*)** as behavior of ***g*(*n*)**

Big-O: Upper Bound Notation

- Let $f(n)$ be a running time function (the efficiency), which we have to evaluate.

- In general a function
  - $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
- Formally
  - $f(n) = O(g(n))$ if $\exists$ positive constants c and n_0 such that $\forall n \geq n_0 : f(n) \leq c \cdot g(n)$

Sorting is O(n^2)

- $f(n) = an^2 + bn + c \rightarrow f(n) = O(n^2)$
- Proof
  - We have to find such c' and n_0 that $\boldsymbol{f(n) \leq c' \cdot g(n)}$ for all $n \geq n_0$
  - If any of a, b, and c are less than 0, replace the constant with its absolute value

$$an^2 + bn + c \leq (a+b+c)n^2 + (a+b+c)n + (a+b+c)$$
$$\leq 3(a+b+c)n^2 \text{ for } n \geq 1$$

Let $c' = 3(a+b+c)$ and let $n_0 = 1 \rightarrow f(n) \leq c'n^2$

Big O Fact

- A polynomial of degree k is $O(n^k)$
- Proof
  - Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$
    - Let $a_i = |b_i|$
  - $f(n) \le a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \le$

$$\le n^k \sum_{i=0}^{k} a_i \frac{n^i}{n^k} \le n^k \sum_{i=0}^{k} a_i \le c n^k$$

Multiply n^k to both denominator and numerator

$\Omega(n)$: Lower Bound Notation

- Let $f(n)$ be a running time function (the efficiency), which we have to evaluate.

- In general a function
  - $f(n)$ is $\Omega(g(n))$ if there exist positive constants c and n_0 such that $f(n) \geq c \cdot g(n) \geq 0$ for all $n \geq n_0$
- Formally
  - $f(n) = \Omega(g(n))$ if $\exists$ positive constants c and n_0 such that $^{\forall}n \geq n_0 : f(n) \geq c \cdot g(n) \geq 0$

$\Omega(n)$: Examples

- Example 1:
  - Suppose running time is $f(n)=an + b$
    - Assume a and b are positive (if not, we may replace them by their absolute values):
      $an + b \geq an \rightarrow f(n) = \Omega(n)$, $c = a$, $n_0 = 1$.
- Example 2: Insertion is $\Omega(n^2)$
  - $f(n)= an^2 + bn + c \rightarrow an^2 + bn + c \geq an^2 \rightarrow$
    $\rightarrow c' = a$; $n_0 = 1 \rightarrow f(n) \geq c'n^2$

$\Theta(n)$: Asymptotic Tight Bound

- A function $f(n)$ is $\Theta(g(n))$ if $\exists$ positive constants c_1, c_2 and n_0 such that

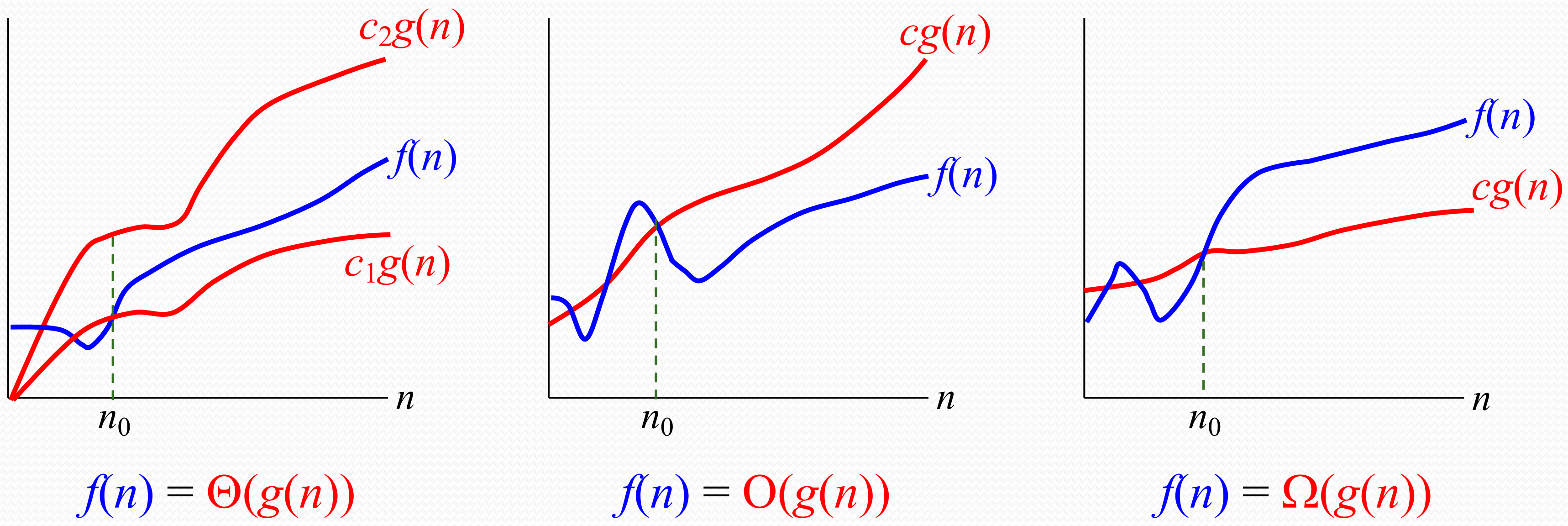
$$c_1\, g(n) \le f(n) \le c_2\, g(n),\ {}^{\forall} n \ge n_0$$

- **Theorem**
  - For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

$\Theta(n)$: Examples

- $f(n) = an^2 + bn + c$, $^{\exists}a$, b and c $a > 0 \rightarrow f(n) = \Theta(n^2)$
- Proof
  - Asymtotic upper bouund: $an^2 + bn + c = O(n^2)$
  - Asymtotic lower bouund: $an^2 + bn + c = \Omega(n^2)$

Graphic Examples of Θ, O, and Ω notations



n_o : the minimum possible value

Other Asymptotic Notations

- A function $f(n)$ is $o(g(n))$ if $\exists$ positive constants c and n_0 such that
  $$f(n) < c \cdot g(n),\ \forall n \geq n_0$$
  We tell that $f(n)$ is asymptotically smaller than $g(n)$

- A function $f(n)$ is $\omega(g(n))$ if $\exists$ positive constants c and n_0 such that
  $$c \cdot g(n) < f(n),\ \forall n \geq n_0$$
  We tell that $f(n)$ is asymptotically larger than $g(n)$

Philosophical Sense

o() is like $<$	ω() is like $>$	
O() is like $\leq$	Ω() is like $\geq$	Θ() is like $=$

- $f(n)$ does not asymptotically exceed $g(n)$ if $f(n) = O(g(n))$
- $f(n)$ asymptotically exceeds $g(n)$ if $f(n) = \Omega(g(n))$
- $f(n)$ is asymptotically smaller than $g(n)$ if $f(n) = o(g(n))$
- $f(n)$ is asymptotically larger than $g(n)$ if $f(n) = \omega(g(n))$

Some properties

- **Transitivity**:
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$
  - Same is true for O, Ω, o, and ω
- **Reflexivity**:
  - $f(n) = \Theta(f(n))$, same is true for O, Ω, o, and ω
- **Symmetry**:
  - $f(n) = \Theta(\mathrm{g}(n))$ if and only if $\mathrm{g}(n) = \Theta(f(n))$
- **Transpose symmetry**:
  - $f(n) = O(\mathrm{g}(n))$ if and only if $\mathrm{g}(n) = \Omega(f(n))$
  - $f(n) = o(\mathrm{g}(n))$ if and only if $\mathrm{g}(n) = \omega(f(n))$

Standard notations and common functions

Monotonicity

- $f(n)$ is *monotonically increasing*
  if $m \leq n$ implies $f(m) \leq f(n)$
- $f(n)$ is *monotonically decreasing*
  if $m \leq n$ implies $f(m) \geq f(n)$
- $f(n)$ is *strictly increasing*
  if $m < n$ implies $f(m) < f(n)$
- $f(n)$ is *strictly decreasing*
  if $m < n$ implies $f(m) > f(n)$

Exponentials

- For all real $a > 0$, m, and n, we have the following identities:
  - $a^0 = 1$
  - $a^1 = a$
  - $a^{-1} = 1/a$
  - $(a^m)^n = a^{mn}$
  - $(a^m)^n = (a^n)^m$
  - $a^m a^n = a^{m+n}$

Exponentials

- For all real constants a and b such that $a > 1$:

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0$$

  from which we can conclude that $n^b = o(a^n)$

  - Any exponential function (a^n) with a base strictly greater than 1 grows faster than any polynomial function (n^b)

- For all real x: $e^x \geq 1 + x$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

  - As x gets closer to 0, e^x gets closer to $1 + x$
  - Equality holds only when $x = 0$

Logarithms

- We shall use the following notations:
  - $\lg n = \log_2 n$ (binary logarithm)
  - $\ln n = \log_e n$ (natural logarithm)
  - $\lg^k n = (\lg n)^k$ (exponentiation)
  - $\lg \lg n = \lg(\lg n)$ (composition)

Logarithms

- For all real $a > 0$, $b > 0$, $c > 0$, and n
  - $a = b^{\log_b a}$
  - $\log_c(ab) = \log_c a + \log_c b$
  - $\log_b a^n = n \log_b a$
  - $\log_b a = \dfrac{\log_c a}{\log_c b} = \dfrac{\ln a}{\ln b}$
  - $\log_b(1/a) = -\log_b a$
  - $\log_b a = \dfrac{1}{\log_a b}$
  - $a^{\log_b c} = c^{\log_b a}$

where, in each equation above, logarithm bases are not 1.

Logarithms

- **Base** of a logarithm
  - Changing the base of a logarithm from one constant to another only changes the value by a constant factor, so we usually don't worry about logarithm bases in asymptotic notation.
  - Convention is to use **lg** (binary logarithm), unless the base actually matters

Logarithms

- Polynomials grow more slowly than exponentials:

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0 \Rightarrow n^b = o(a^n)$$

- Logarithms grow more slowly than polynomials (substituting $n \to \lg n$, $a \to 2^a$)

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0$$

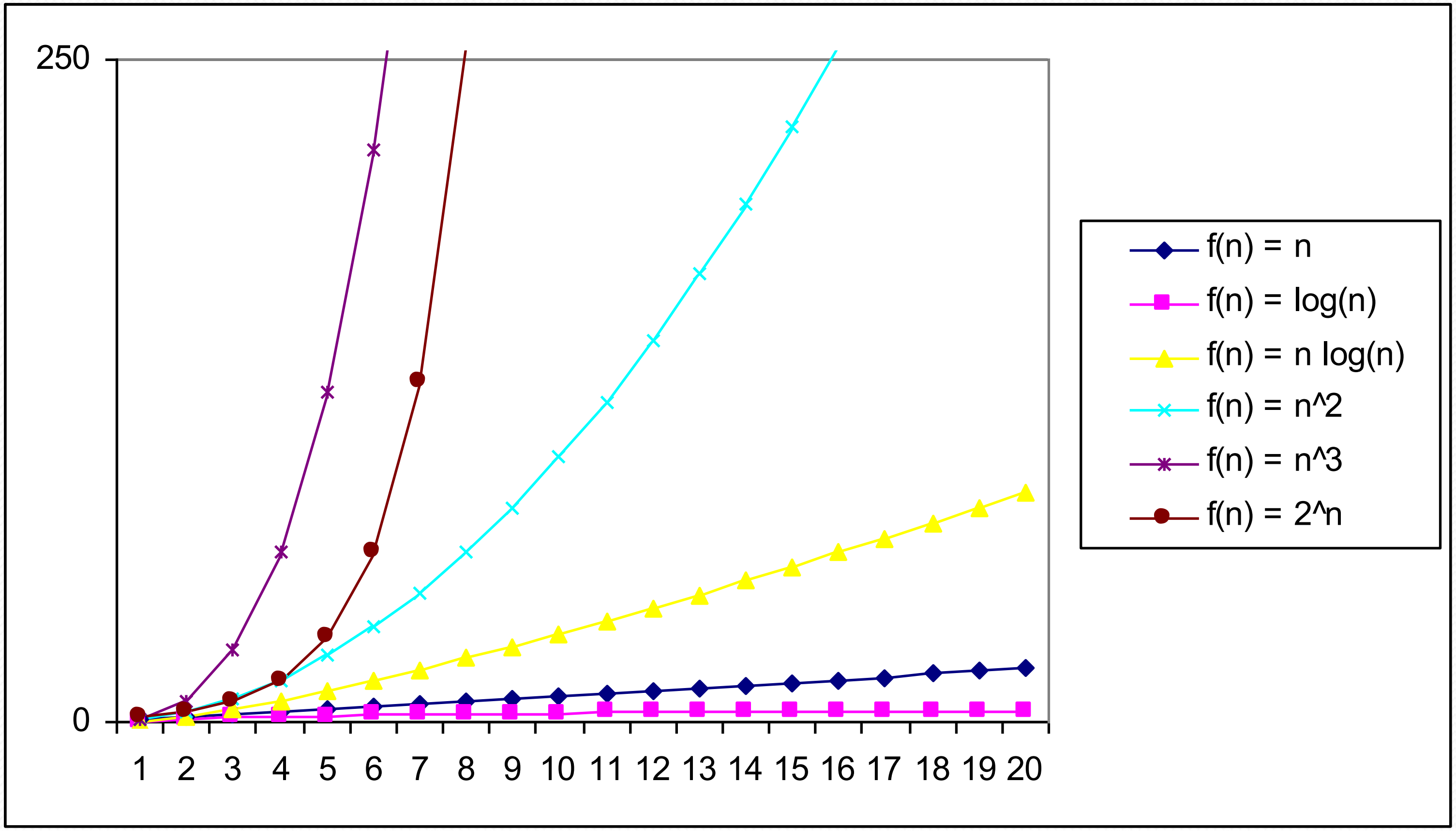
from which we can conclude that $\mathbf{lg}^{b}\, n = \mathbf{o}(n^a)$

  - For any constant $a > 0$, any positive polynomial function grows faster than any polylogarithmic function.

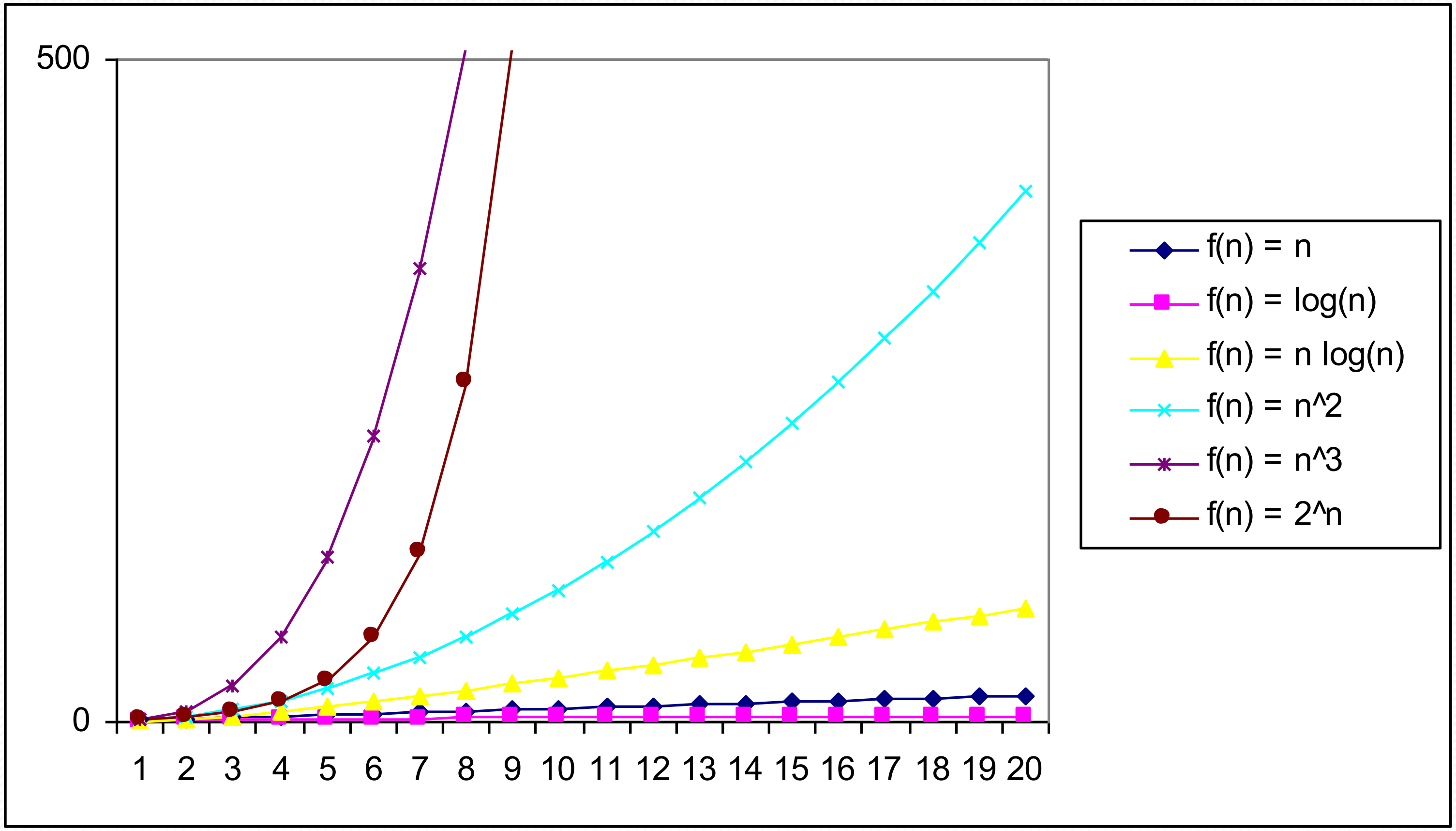
Floors and ceilings

- For any real number x:
  - $\lfloor x \rfloor$ is the greatest integer less than or equal to x ("the floor of x")
  - $\lceil x \rceil$ is the least integer greater than or equal to x ("the ceiling of x")
  - Both functions $f(x) = \lfloor x \rfloor$ and $f(x) = \lceil x \rceil$ are monotonically increasing
  - For any real x: $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
  - For any integer n: $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$

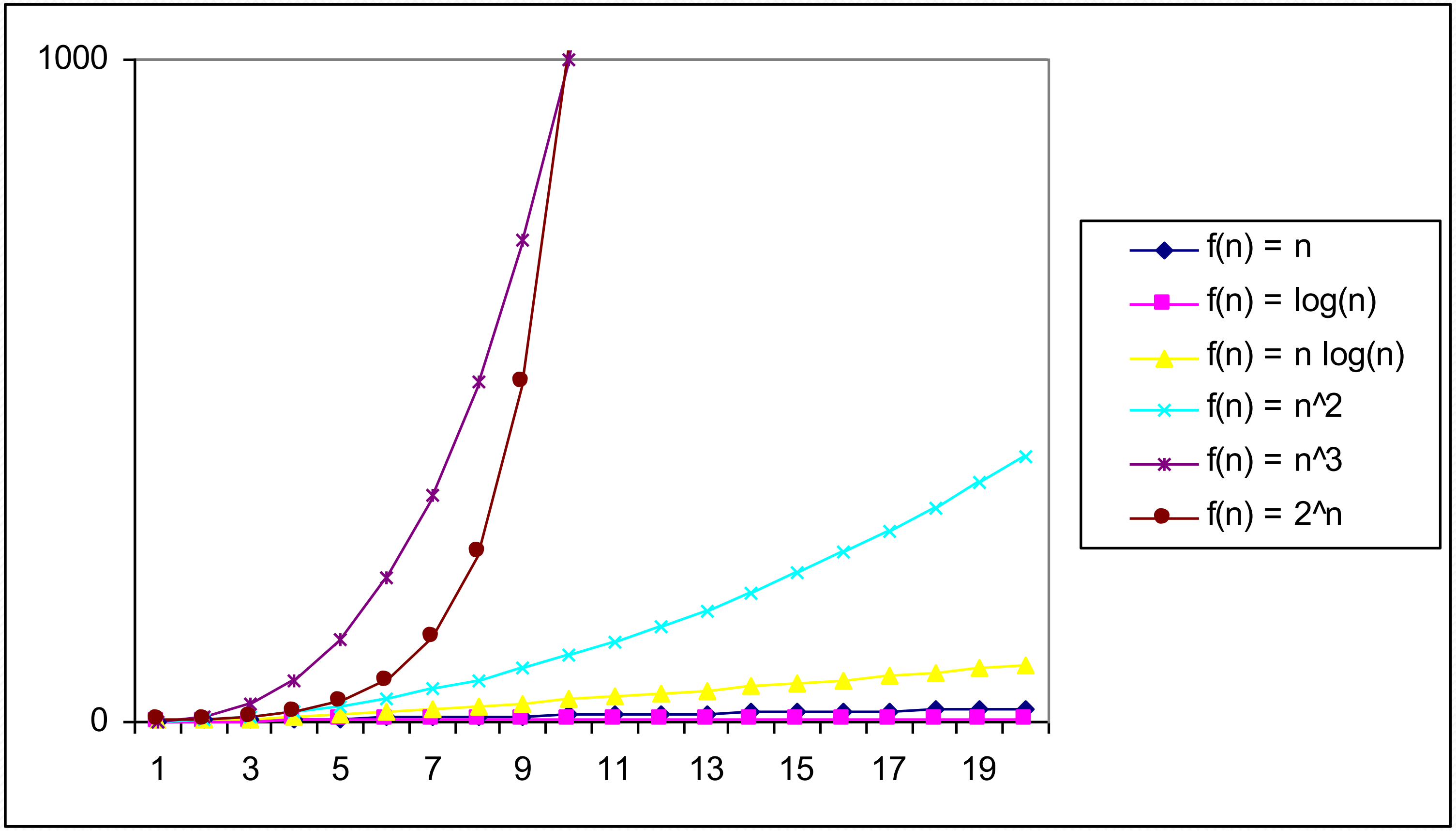
Practical Complexity



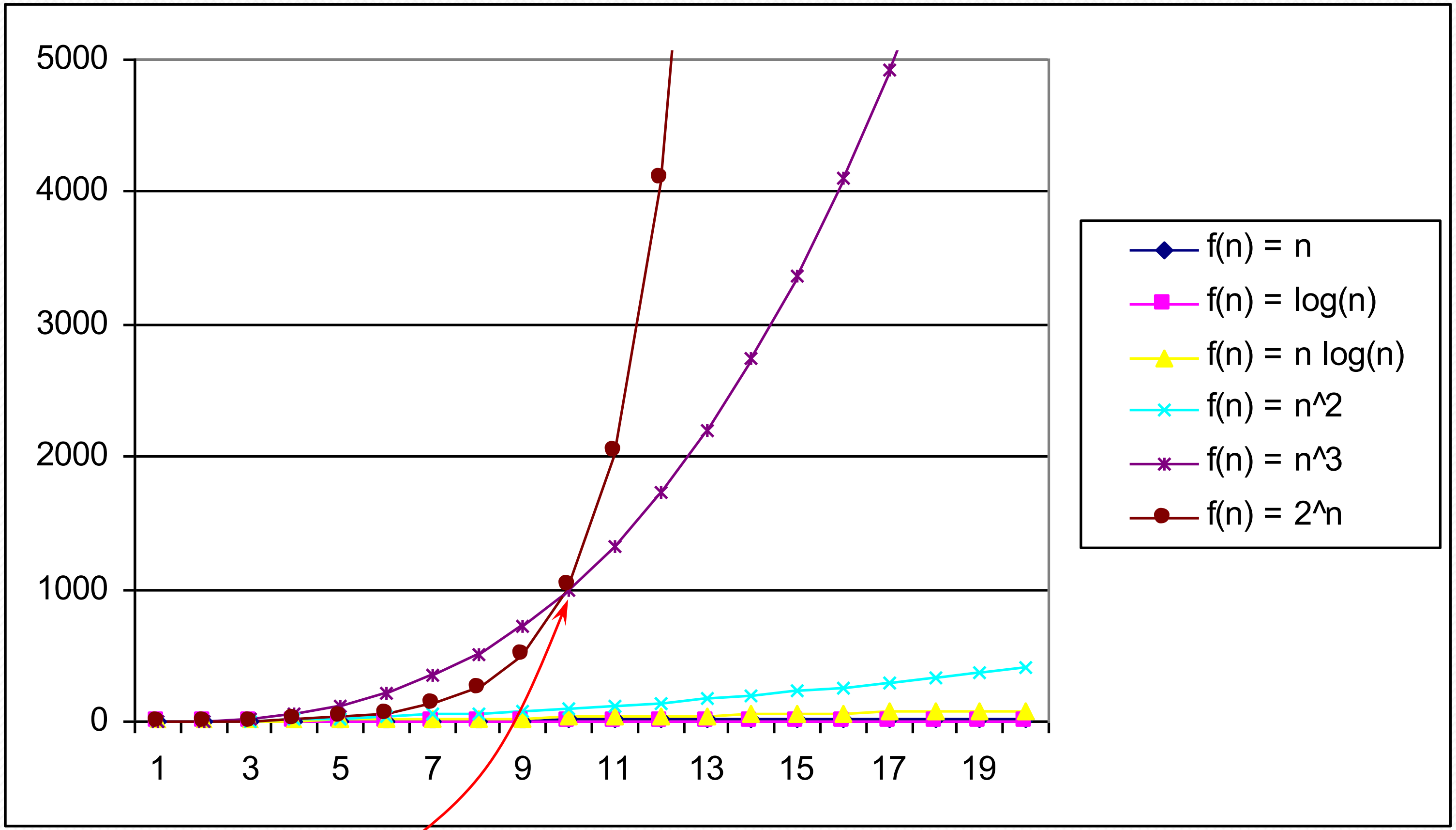
Practical Complexity



Practical Complexity



Practical Complexity



2^n growths faster than n^3 starting from n=10