CSE 2215: Data Structure and Algorithms-I

Introduction, Complexity Analysis

The Course

- Text Books
  - *Introduction to Algorithms* (Third edition)
    Cormen, Leiserson, Rivest, and Stein
    - An excellent reference you should own
  - *Data Structures and Algorithms in C++*
    Goodrich, Tamassia, and Mount
- Instructor: Dr. Md. Abul Kashem Mia
  - Professor, CSE Dept and Pro-Vice Chancellor, UIU
  - *kashem@uiu.ac.bd*

What is a Data Structure?

- Data is a collection of facts, such as values, numbers, words, measurements, or observations.
- Structure means a set of rules that holds the data together.
- A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.
  - Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.
  - Data structures provide a means to manage huge amount of data efficiently.
  - Usually, efficient data structures are a key to designing efficient algorithms.
  - Data structures can be nested.

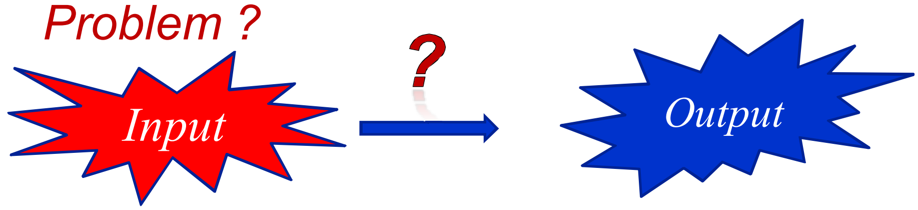
Types of Data Structures

- Data structures are classified as either
  - Linear (*e.g*, arrays, linked lists), or
  - Nonlinear (*e.g*, trees, graphs, etc.)
- A data structure is said to be linear if it satisfies the following four conditions
  - There is a unique element called the first
  - There is a unique element called the last
  - Every element, except the last, has a unique successor
  - Every element, except the first, has a unique predecessor
- There are two ways of representing a linear data structure in memory
  - By means of sequential memory locations (arrays)
  - By means of pointers or links (linked lists)

What is an Algorithm?

An algorithm is a sequence of instructions that one must perform in order to solve a well-formulated problem.

Problem ?



Algorithms did not start with computers; they have been with us from ancient times; nor they are limited to computer science.

What is an Algorithm?

- An algorithm is a sequence of computational steps that transform the input into the output.
- Algorithms are the ideas behind computer programs.
- An algorithm is a tool for solving a well-specified computational problem.
  - An algorithm is said to be correct if, for every input instance, it halts with the correct output.
  - An incorrect algorithm might not halt at all on some input instances, or it might halt with other than the desired output.

Define a Problem, and Solve It

- **Problem**:
  - Description of Input-Output relationship
- **Algorithm**:
  - A sequence of computational steps that transform the input into the output.
- **Data Structure:**
  - An organized method of storing and retrieving data.
- **Our Task:**
  - Given a problem, ***design a correct and good algorithm*** that solves it.
  - To design a good algorithm, find a suitable data structure.

Why Study Data Structures and Algorithms

- You will be able to write better, faster and more elegant code.
- You will be able to think more clearly, more abstractly and more mathematically.
- You will be able to solve new problems.
- You will be able to give non-trivial methods to solve problems.
- You will improve your research skills in almost any areas.
- It's one of the most challenging and interesting area of Computer Science.

Why Study Data Structures and Algorithms

- Almost all big companies want programmers with knowledge of algorithms: Microsoft, Apple, Google, Facebook, Oracle, IBM, Yahoo, etc.
- In the most job interviews, they will ask you several questions about algorithms and/or data structures. They may even ask you to write pseudo or real code on the spot.
- Your knowledge of algorithms will set you apart from the masses of interviewees who know only how to program.
- If you want to start your own company, you should know that many startups are successful because they have found better algorithms for solving a problem.

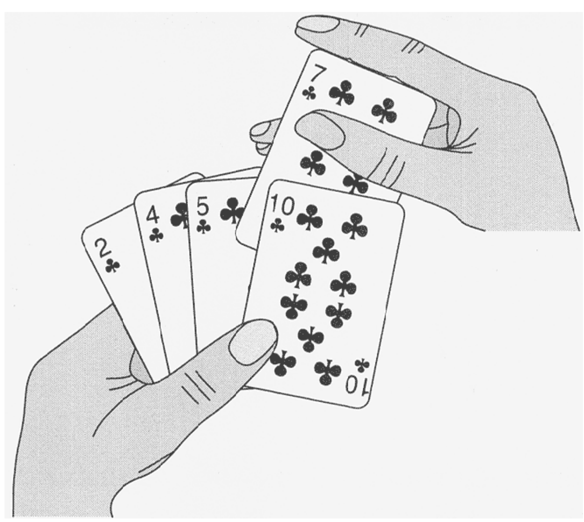
What do we Analyze of an Algorithm?

- Correctness
  - Does the input/output relation match algorithm requirement?
- Amount of work done (complexity)
  - Basic operations to do task
- Amount of space used
  - Memory used
- Simplicity, clarity
  - Verification and implementation.
- Optimality
  - Is it impossible to do better?

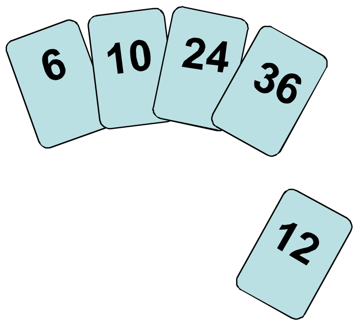
Running Time

- Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
    - y = m * x + b
    - c = 5 / 9 * (t - 32)
    - z = f(x) + g(y)
- We can be more exact if need to be

An Example: Insertion Sort

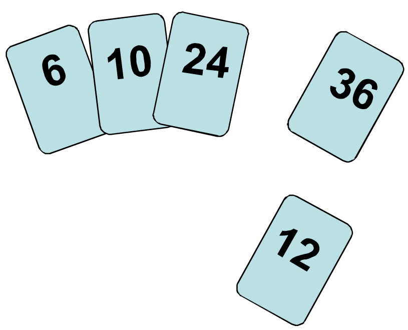


An Example: Insertion Sort



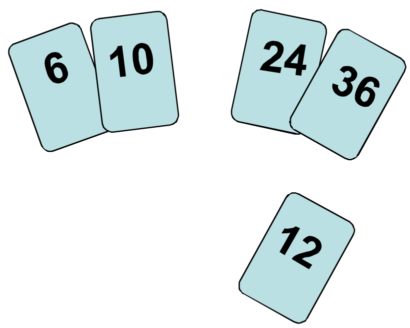
To insert 12, we need to make room for it by moving first 36 and then 24.

An Example: Insertion Sort



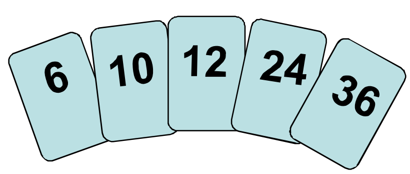
To insert 12, we need to make room for it by moving first 36 and then 24.

An Example: Insertion Sort



To insert 12, we need to make room for it by moving first 36 and then 24.

An Example: Insertion Sort



To insert 12, we need to make room for it by moving first 36 and then 24.

An Example: Insertion Sort

input array

5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:

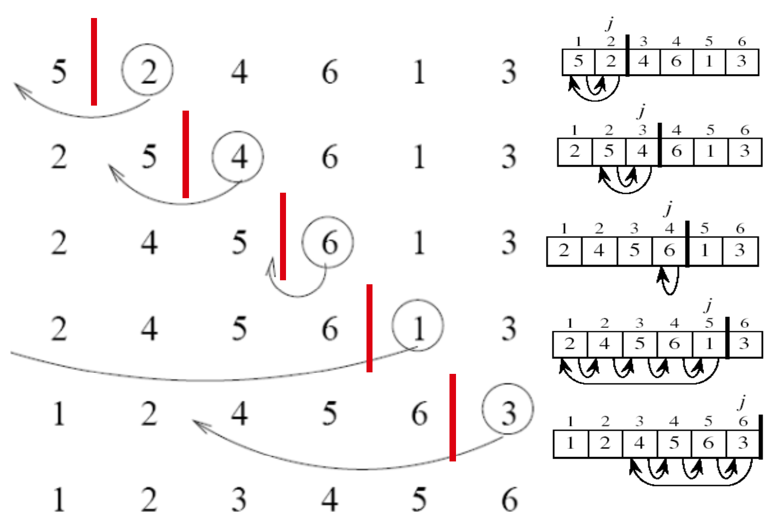
left sub-array *right sub-array*

2 5 | 4 6 1 3

sorted *unsorted*

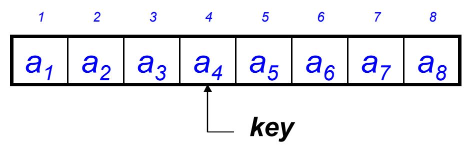
17

An Example: Insertion Sort



18

Insertion Sort



Alg.: INSERTION-SORT(A)

for j ← 2 **to** n
    do key ← A[j]
      ▷Insert A[j] into the sorted sequence A[1 . . j -1]
      i ← j - 1
      while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
              i ← i - 1
      A[i + 1] ← key

- Insertion sort – sorts the elements in place

19

Time Analysis of Insertion Sort

INSERTION-SORT(*A*)	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
▷Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

t_j: # of times the while statement is executed at iteration j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

20

Best Case Analysis

- The array is already sorted *"**while** i > 0 and A[i] > key"*
  - A[i] ≤ key upon the first time the **while** loop test is run (when $i = j - 1$)
  - $t_j = 1$
- T(n) = $c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$
  $= (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$
  $= an + b = O(n)$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

21

Worst Case Analysis

- The array is in reverse sorted order *"**while** i > 0 and A[i] > key"*
  - Always A[i] > key in **while** loop test
  - Have to compare key with all elements to the left of the j-th position $\Rightarrow$ compare with j-1 elements $\Rightarrow t_j = j$

using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \Rightarrow \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \Rightarrow \sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$ *we have:*

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

$$= an^2 + bn + c$$

- $T(n) = O(n^2)$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

22

Analyzing Some Algorithms

n ← length[A]

min ←1

for i ← 1 **to** n **do**

  if A[i] < A[min] **then**

    min ← i

Analyzing Some Algorithms

n ← length[A]

min ←1

for i ← 1 **to** n **do**

  for j ← 1 **to** n **do**

    if A[j] < A[min] **then**

      min ← j

Analyzing Some Algorithms

n ← length[A]

min ←1

for i ← 1 **to** n **do**

  for j ← i +1 **to** n **do**

    if A[j] < A[min] **then**

      min ← j

Analyzing Some Algorithms

n ← length[A]

min ←1

for i ← 1 **to** n **do**

  for j ← n **downto** i +1 **do**

    if A[j] < A[min] **then**

      min ← j

Analyzing Some Algorithms

Analyze both *best* and *worst* case time-complexities of the following algorithm.

```
n  ← length[A]
m ← length[B]
for i ← 1 to n – 1 do
    min ← i
    for j ← 1 to m do
        if (A[ i ] + B[ j ] >= 10) then
            break
    print A[min]
```


Analyzing Some Algorithms

Analyze both *best* and *worst* case time-complexities of the following algorithm.

```
n ← length[A]
for j ← 1 to n – 1 do
    min ← j
    for i ← j + 1 to n do
        for k ← 1 to n do
            if A[ i ] < A[min] then
                break;
```

Asymptotic Performance

- We care most about *asymptotic performance*
  - How does the algorithm behave as the problem size gets very large?
    - Running time
    - Memory/storage requirements
    - Bandwidth/power requirements/logic gates/etc.

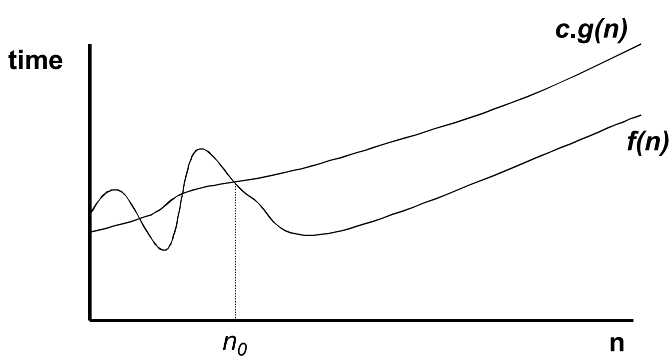
Asymptotic Analysis

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee
- Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is “average”?
    - Random (equally likely) inputs
    - Real-life inputs
- Best case

Upper Bound Notation

- We say InsertionSort's run time is $O(n^2)$
  - Properly we should say run time is *in* $O(n^2)$
  - Read O as "Big-O" (you'll also hear it as "order")
- In general a function
  - $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$
- Formally
  - $O(g(n)) = \{ f(n)$: $\exists$ positive constants c and n_0 such that $0 \le f(n) \le c \cdot g(n)\ \forall\ n \ge n_0 \}$

Upper Bound Notation



$O(g(n)) = \{ f(n)$: $\exists$ positive constants c and n_0 such that $0 \le f(n) \le c \cdot g(n)\ \forall\ n \ge n_0 \}$

We say $g(n)$ is an *asymptotic upper bound* for $f(n)$

Upper Bound Notation

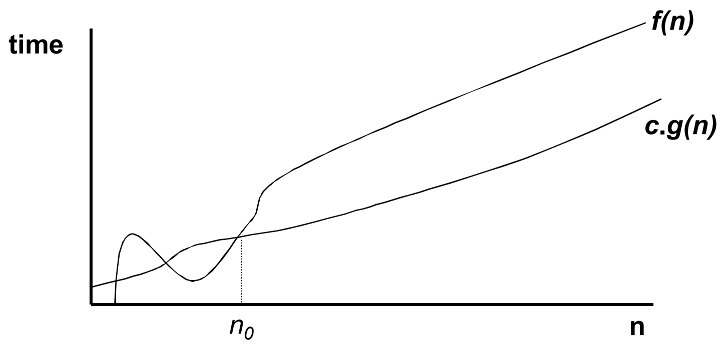
Upper Bound Notation

Upper Bound Notation

Lower Bound Notation

- We say InsertionSort's run time is $\Omega(n)$
- In general a function
  - $f(n)$ is $\Omega(g(n))$ if $\exists$ positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n) \quad \forall\ n \geq n_0$
- Proof:
  - Suppose run time is an + b
    - Assume a and b are positive
  - an $\leq$ an + b

Lower Bound Notation

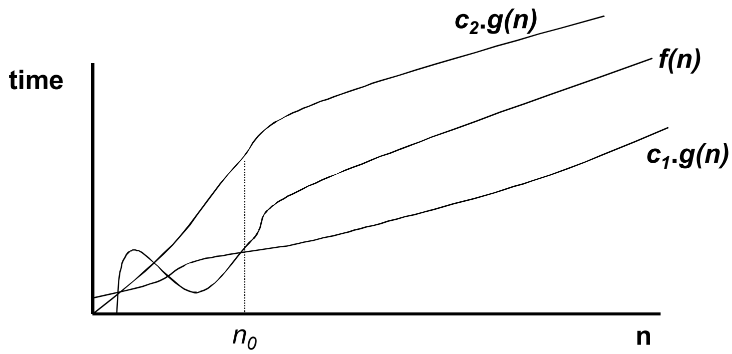


$\Omega(g(n)) = \{ f(n)$: $\exists$ positive constants c and n_0 such that
$0 \le c \cdot g(n) \le f(n) \; \forall \; n \ge n_0 \}$

We say $g(n)$ is an *asymptotic lower bound* for $f(n)$

Lower Bound Notation

Asymptotic Tight Bound



- A function $f(n)$ is $\Theta(g(n))$ if $\exists$ positive constants c_1, c_2, and n_0 such that $0 \le c_1\ g(n) \le f(n) \le c_2\ g(n)\ \forall\ n \ge n_0$
- Theorem: $f(n)$ is $\Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

We say $g(n)$ is an *asymptotic tight bound* for $f(n)$

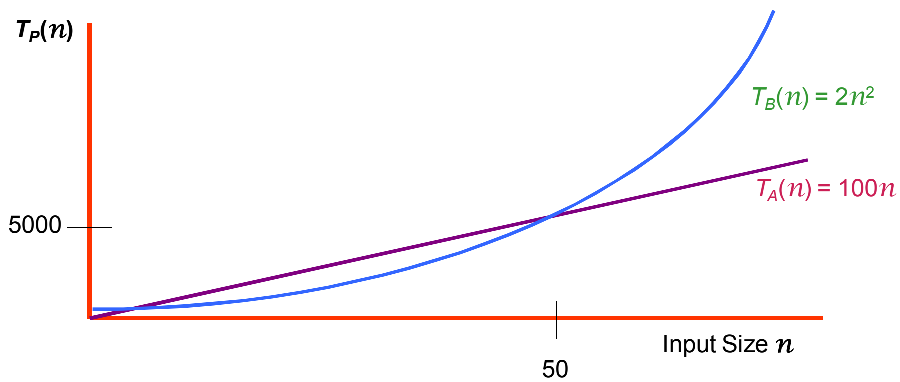
Asymptotic Tight Bound

Practical Complexity

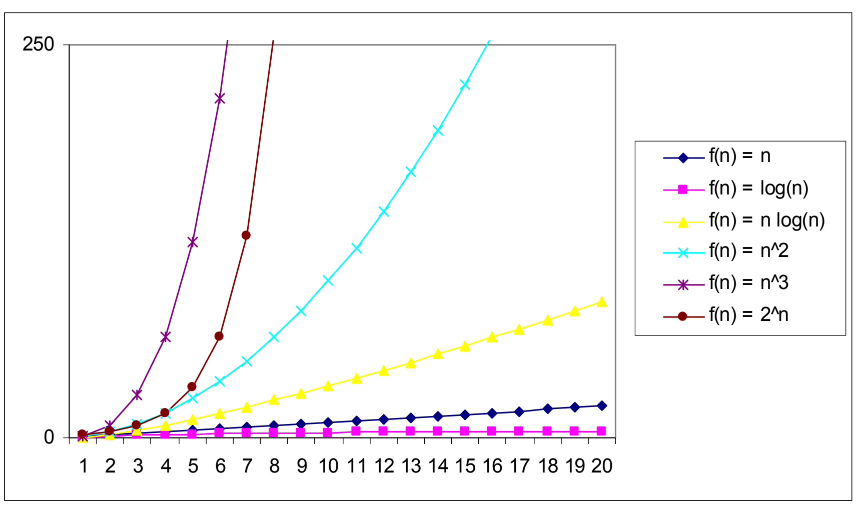
For large input sizes, constant terms are insignificant

Program A with running time $T_A(n)= 100n$

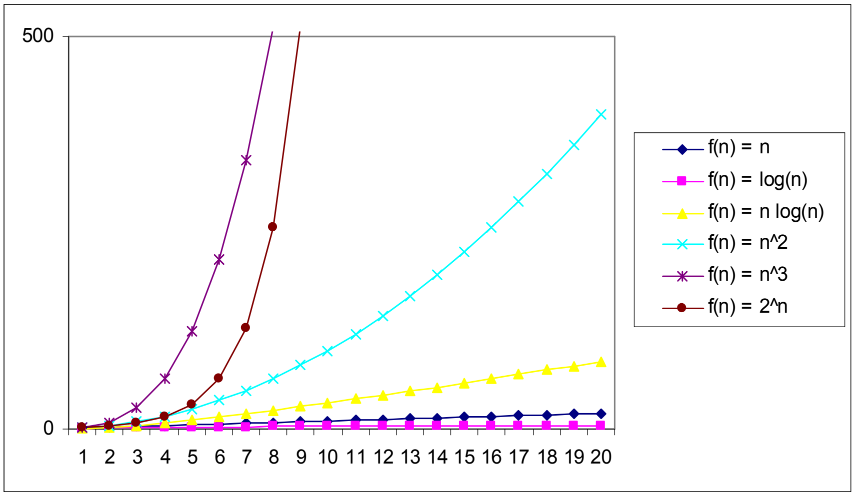
Program B with running time $T_B(n)= 2n^2$



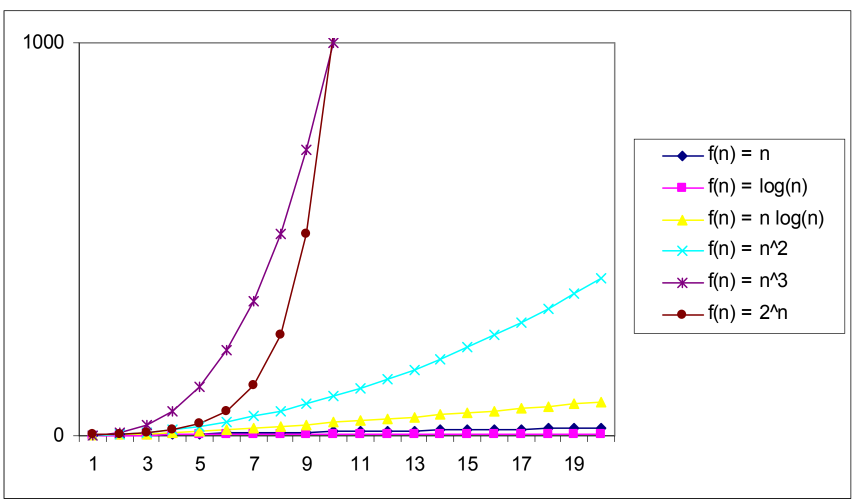
Practical Complexity



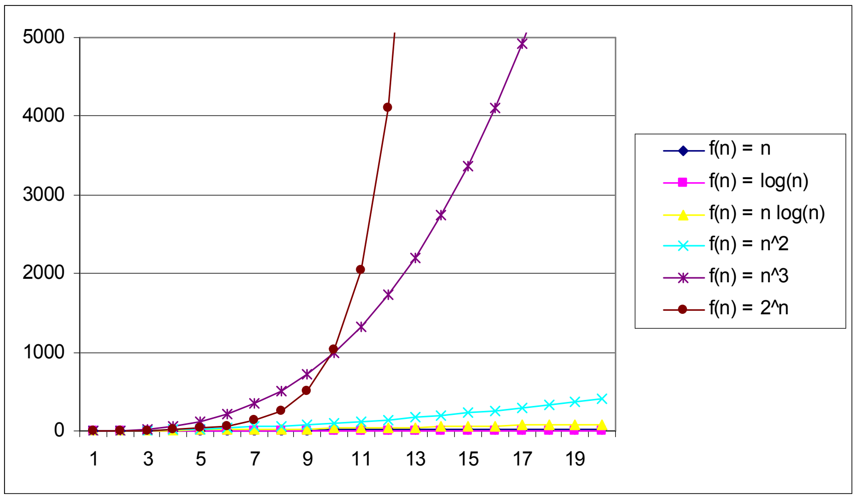
Practical Complexity



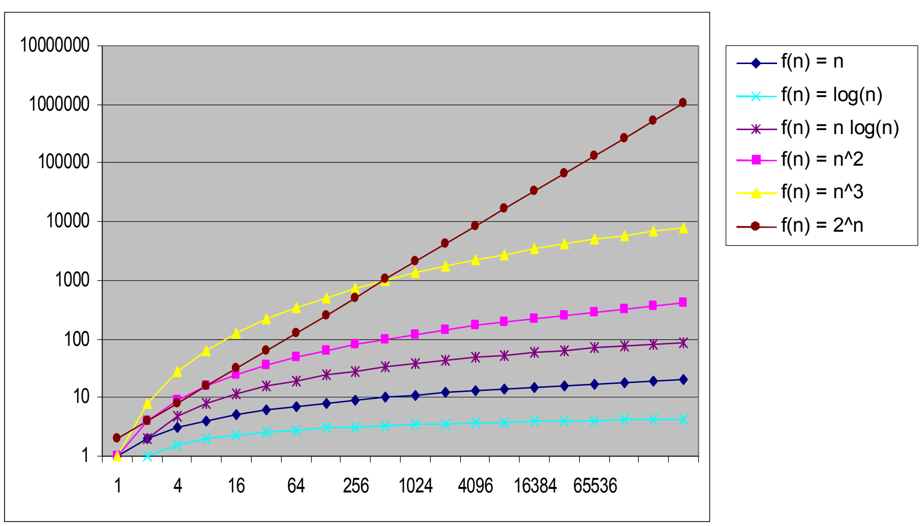
Practical Complexity



Practical Complexity



Practical Complexity



Practical Complexity

Function	Descriptor	Big-Oh
c	Constant	$O(1)$
$\log n$	Logarithmic	$O(\log n)$
n	Linear	$O(n)$
$n \log n$	$n \log n$	$O(n \log n)$
n^2	Quadratic	$O(n^2)$
n^3	Cubic	$O(n^3)$
n^k	Polynomial	$O(n^k)$
2^n	Exponential	$O(2^n)$
$n!$	Factorial	$O(n!)$

Other Asymptotic Notations

- A function $f(n)$ is $o(g(n))$ if $\exists$ positive constants c and n_0 such that
  $$f(n) < c\, g(n) \ \forall\ n \geq n_0$$
- A function $f(n)$ is $\omega(g(n))$ if $\exists$ positive constants c and n_0 such that
  $$c\, g(n) < f(n) \ \forall\ n \geq n_0$$
- Intuitively,
  - $o(\,)$ is like $<$
  - $O(\,)$ is like $\leq$
  - $\omega(\,)$ is like $>$
  - $\Omega(\,)$ is like $\geq$
  - $\Theta(\,)$ is like $=$