DATA STRUCTURES AND ALGORITHMS

Textbook:

Fundamentals of Data Structure in C++, Silicon Press

Instructor: Yun Wang

School of Computer Science & Engineering

Southeast University

ywang_cse@seu.edu.cn

Teaching assistants:

沈文秀：473084676@qq.com

Total class hours: 64

week 1-16

Total lab. hours: 16

Assignments and projects:

- Should be handed to teaching assistants.

References:

金远平，数据结构（C++描述），清华大学出版社，2005

殷人昆主编，数据结构（第2版），清华大学出版社，2007

Prerequisites:

Programming Language: C, C++

Evaluation:

Continuous Assessment:	**10%**
Class Discussion :	**20%**
Exercises and Projects:	**30%**
Final Examination :	**50%**

Tips

- Make good use of your time in class
  - Listening
  - Thinking
  - Taking notes
- Expend your free time
  - Go over
  - Programing
- Take a pen and some paper with you
  - Notes
  - Exercises

Course Overview

In Computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

- Basic Concepts
- Arrays
- Stack and Queue
- Linked Lists
- Trees
- Graphs
- Sorting
- Hashing
- Search Trees

1 Basic Concepts

- System Life Cycle
- Data Abstraction and Encapsulation
- Algorithm Specification
- Performance Analysis and Measurement

1 Basic Concepts

- System Life Cycle
- Data Abstraction and Encapsulation
- Algorithm Specification
- Performance Analysis and Measurement

Sorting

- Rearrange a[0], a[1], …, a[n-1] into ascending order. When done, a[0] <= a[1] <= … <= a[n-1]
- 8, 6, 9, 4, 3 => 3, 4, 6, 8, 9

Sort Methods

- **Insertion Sort**
- Bubble Sort
- Selection Sort
- Counting Sort
- Shell Sort
- Heap Sort
- Merge Sort
- Quick Sort
- ……

Insert An Element

- Given a sorted list/sequence, insert a new element
- Given 3, 6, 9, 14
- Insert 5
- Result 3, 5, 6, 9, 14

Insert an Element

- 3, 6, 9, 14   insert 5
- Compare new element (5) and last one (14)
- Shift 14 right to get 3, 6, 9, , 14
- Shift 9 right to get 3, 6, , 9, 14
- Shift 6 right to get 3, , 6, 9, 14
- Insert 5 to get 3, 5, 6, 9, 14

Insert An Element

```
// insert t into a[0:i-1]
int j;
for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];
a[j + 1] = t;
```

Insertion Sort

- Start with a sequence of size 1
- Repeatedly insert remaining elements

Insertion Sort

- Sort 7, 3, 5, 6, 1
- Start with 7 and insert 3 => 3, 7
- Insert 5 => 3, 5, 7
- Insert 6 => 3, 5, 6, 7
- Insert 1 => 1, 3, 5, 6, 7

Insertion Sort

```
for (int i = 1; i < a.length; i++)
{// insert a[i] into a[0:i-1]
   // code to insert comes here
}
```

Insertion Sort

```
for (int i = 1; i < a.length; i++)
{// insert a[i] into a[0:i-1]
   int t = a[i];
   int j;
   for (j = i - 1; j >= 0 && t < a[j]; j--)
       a[j + 1] = a[j];
   a[j + 1] = t;
}
```

Purpose

Provide the tools and techniques necessary to design and implement large-scale software systems

System Life Cycle

(1) Requirements

specifications of purpose

input

output

(2) Analysis

break the problem into manageable pieces

bottom-up

top-down

(3) Design

a SYSTEM

data objects

operations on them

TO DO

abstract data type

algorithm specification and design

(4) Refinement and coding

- representations for data object
- algorithms for operations
- components reuse

(5) Verification and maintenance

- correctness proofs
- testing
- error removal
- update

1 Basic Concepts

- System Life Cycle
- Data Abstraction and Encapsulation
- Algorithm Specification
- Performance Analysis and Measurement

Data Abstraction and Encapsulation

Data Encapsulation or information Hiding is the concealing of the implementation details of a data object from the outside world.

Data Abstraction is the separation between the *specification* of a data object and its *implementation*.

A Data Type is a collection of *objects* and a set of *operations* that act on those objects.

predefined and user-defined:

char, int, arrays, structs, classes.

An Abstract Data Type (ADT) is a data type with the specification of the objects and the specification of the operations on the objects being separated from the representation of the objects and the implementation of the operations.

Benefits of data abstraction and data encapsulation

(1) Simplification of software development

(2) Testing and debugging

(3) Reusability

data structures implemented as distinct entities of a software system

(4) Modifications to the representation of a data type

a change in the internal implementation of a data type will not affect the rest of the program as long as its interface does not change.

1 Basic Concepts

- System Life Cycle
- Data Abstraction and Encapsulation
- Algorithm Specification
- Performance Analysis and Measurement

Algorithm Specification

An algorithm is finite set of instructions that, if followed, accomplishes a particular task.

Must satisfy the following criteria:

(1) Input Zero or more quantities externally supplied.

(2) Output At least one quantity is produced.

(3) Definiteness Clear and unambiguous.

(4) Finiteness Terminates after a finite number of steps.

(5) Effectiveness Basic enough, feasible

Compare: algorithms and programs

Finiteness

Recursion

- Recursive Statement
- Terminating Condition
- Example: n!

1 Basic Concepts

- System Life Cycle
- Data Abstraction and Encapsulation
- Algorithm Specification
- Performance Analysis and Measurement

Performance Analysis and Measurement

Definition:

The Space complexity of a program is the amount of memory it needs to run to completion.

The Time complexity of a program is the amount of computer time it needs to run to completion.

(1) Priori estimates --- Performance analysis

(2) Posteriori testing--- Performance measurement

Performance Analysis

Space complexity

The space requirement of program P:

$$S(P)=c+S_P(\text{instance characteristics})$$

We concentrate solely on S_P.

Performance Analysis

float Rsum (**float** *a, **const int** n) //compute $\sum_{i=0}^{n-1} a[i]$ recursively

{

if (n <=0) **return** 0**;**

else return (Rsum(a,n-1)+a[n-1])**;**

}

- **The instances are characterized by**

  n

- **each call requires 4 words (n, a, return value, return address)**
- **the depth of recursion is**

  $n+1$

- **$S_{rsum}(n) = 4(n+1)$**

Time complexity

Run time of a program P:

$$T(P) = c + t_P(\text{instance characteristics})$$

A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of instance characteristics.

Step Count

- A step is an amount of computing that does not depend on the instance characteristic n
- 10 adds, 100 subtracts, 1000 multiplies can all be counted as a single step
- n adds cannot be counted as 1 step

Our main concern:

how many steps are needed by a program to solve a particular problem instance?

Example 1.12

```
count=0;
float  Rsum (float *a, const int n)
{
      count++; // for if
      if (n <=0) {
         count++; // for return
         return 0;
       }
      else {
         count++; // for return
         return (Rsum(a,n-1)+a[n-1]);
       }
}
```

$t_{Rsum}(0) = 2,$

$$
\begin{aligned}
t_{Rsum}(n) &= 2 + t_{Rsum}(n-1) \\
&= 2+2+ t_{Rsum}(n-2) \\
&\;. \\
&\;. \\
&\;. \\
&= 2n + t_{Rsum}(0) \\
&= 2n+2
\end{aligned}
$$

Sometime, the instance characteristics is related with the content of the input data set.

e.g., *BinarySearch*.

Hence:

- **best-case**
- **worst-case,**
- **average-case.**

Asymptotic Notation

Because of the inexactness of what a step stands for, we are mainly concerned with the magnitude of the number of steps.

Definition [O]: f(n)=O(g(n)) iff there exist positive constants c and n_0 such that $f(n) \leq c\, g(n)$ for all n, $n>n_0$.

Example 1.13: $3n+2=O(n)$, $6*2^n+n^2=O(2^n)$,…

Note g(n) is an upper bound.

$n=O(n^2)$, $n=O(2^n)$, …,

for $f(n)=O(g(n))$ to be informative, g(n) should be

as small as possible.

In practice, the coefficient of g(n) should be 1. We never say O(3n).

Theory 1.2: if $f(n)=a_m n^m+\ldots+ a_1 n+a_0$, then $f(n)=O(n^m)$.

When the complexity of an algorithm is actually, say, O(log n), but we can only show that it is O(n) due to the limitation of our knowledge, it is OK to say so. This is one benefit of O notation as upper bound.

Self-study:

Ω --- low bound

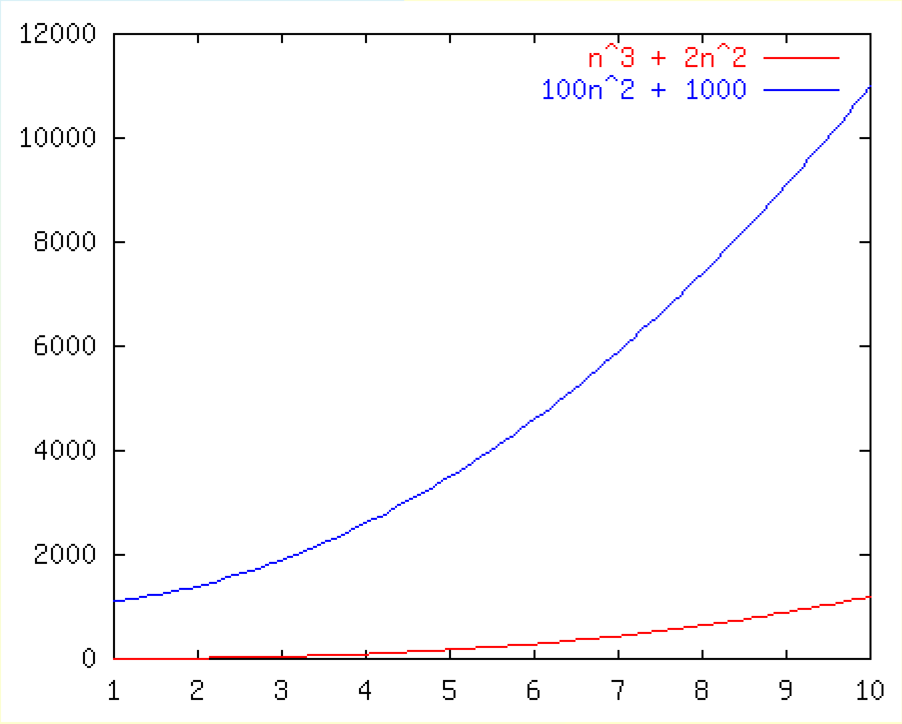
Θ --- equal bound

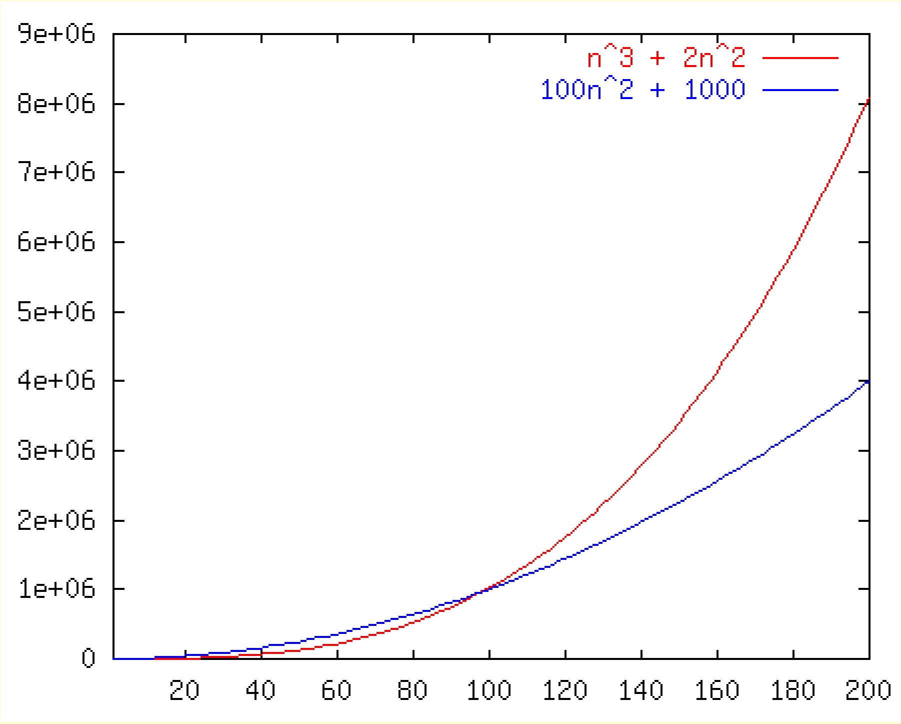
A Few Comparisons

Function #1		Function #2
$n^3 + 2n^2$	$\longleftrightarrow$	$100n^2 + 1000$
$n^{0.1}$	$\longleftrightarrow$	$\log n$
$n + 100n^{0.1}$	$\longleftrightarrow$	$2n + 10 \log n$
$5n^5$	$\longleftrightarrow$	$n!$
$n^{-15}2^n/100$	$\longleftrightarrow$	$1000n^{15}$
$8^{2\log n}$	$\longleftrightarrow$	$3n^7 + 7n$

Race I

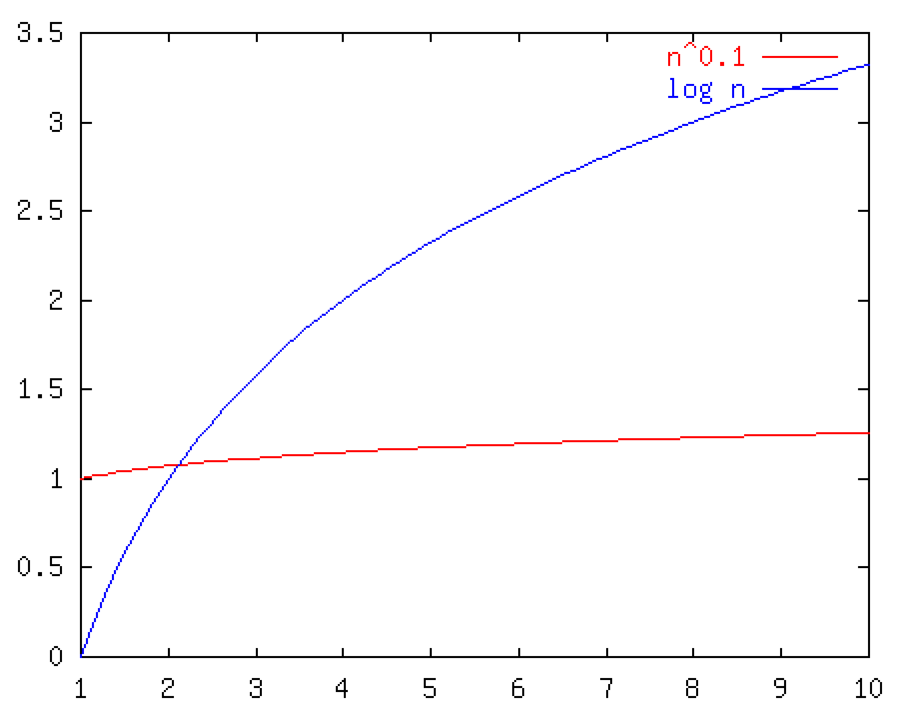
$n^3 + 2n^2$ vs. $100n^2 + 1000$

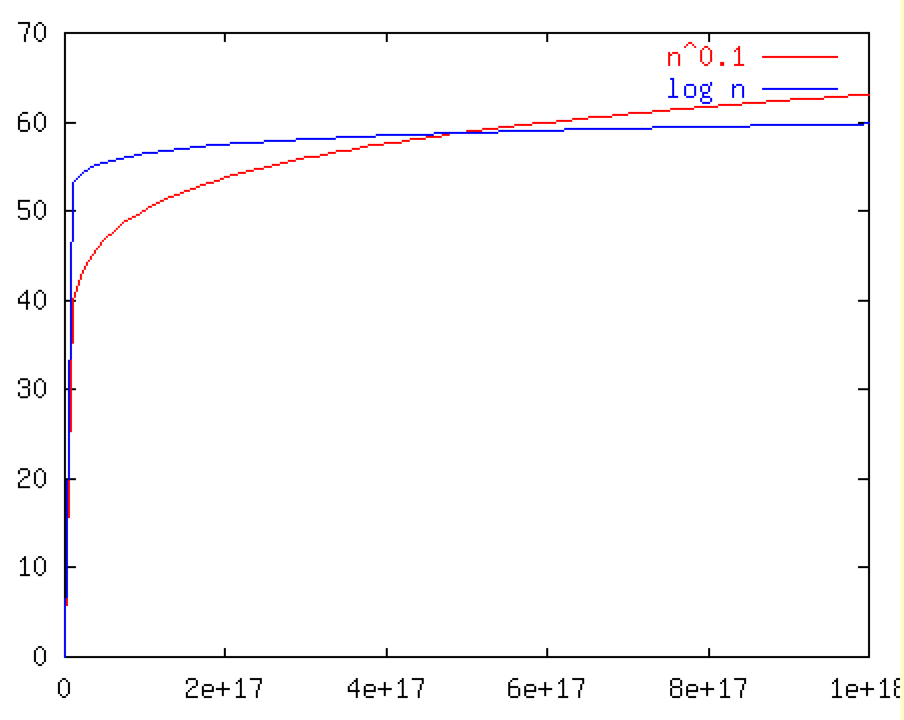




Race II

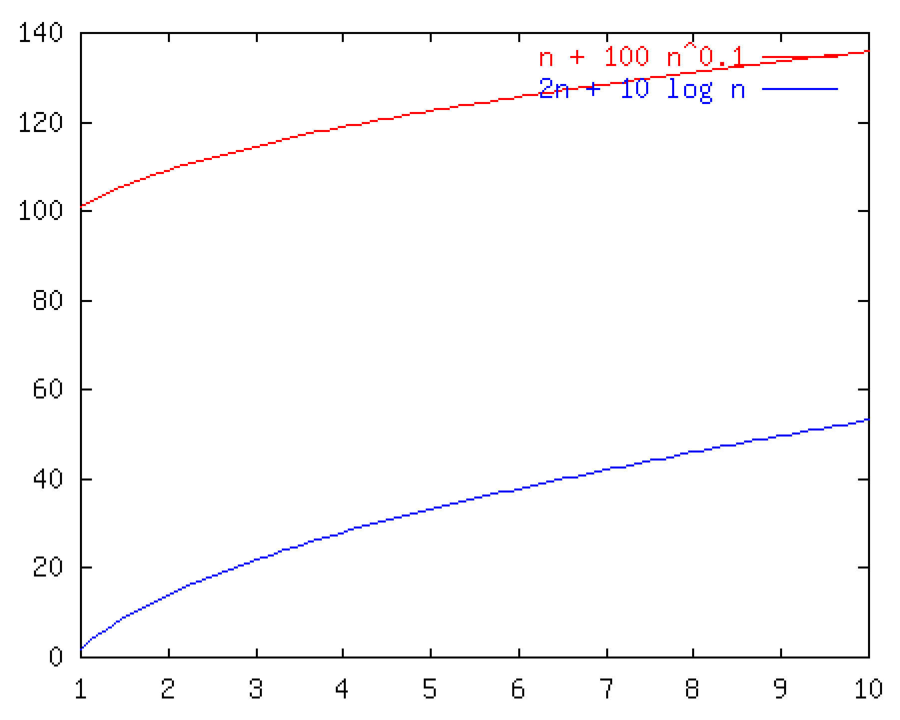
$n^{0.1}$ vs. $\log n$

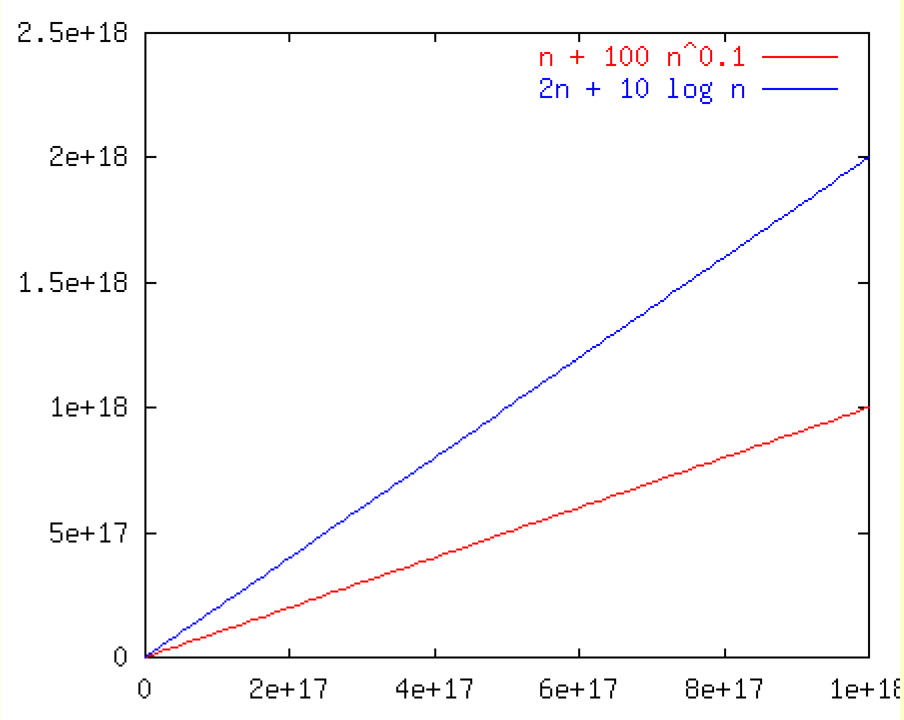




Race III

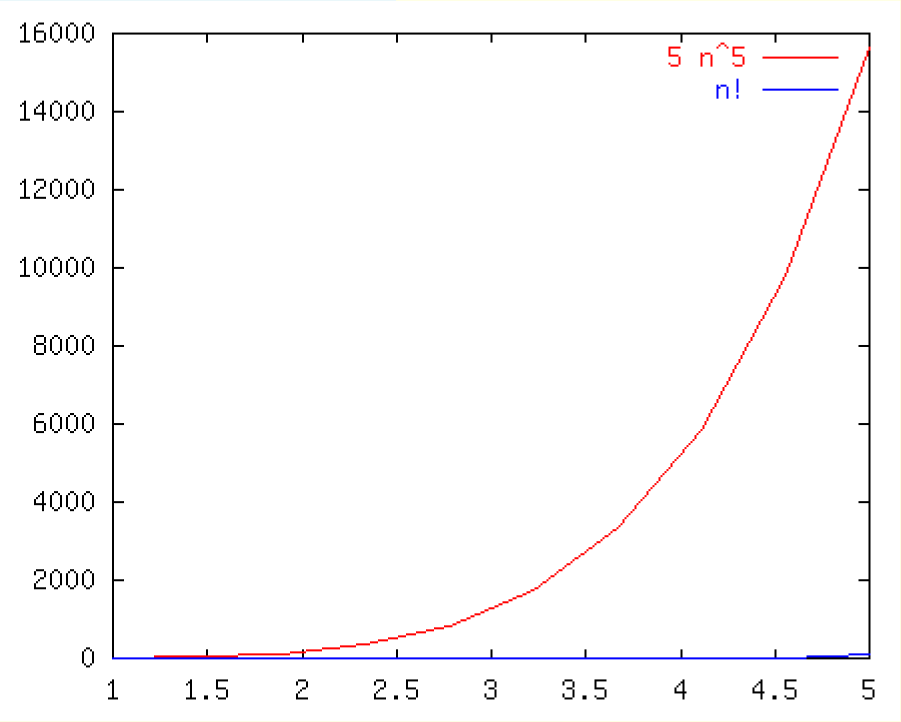
$\mathbf{n + 100n^{0.1}}$ vs. $\mathbf{2n + 10 \log n}$

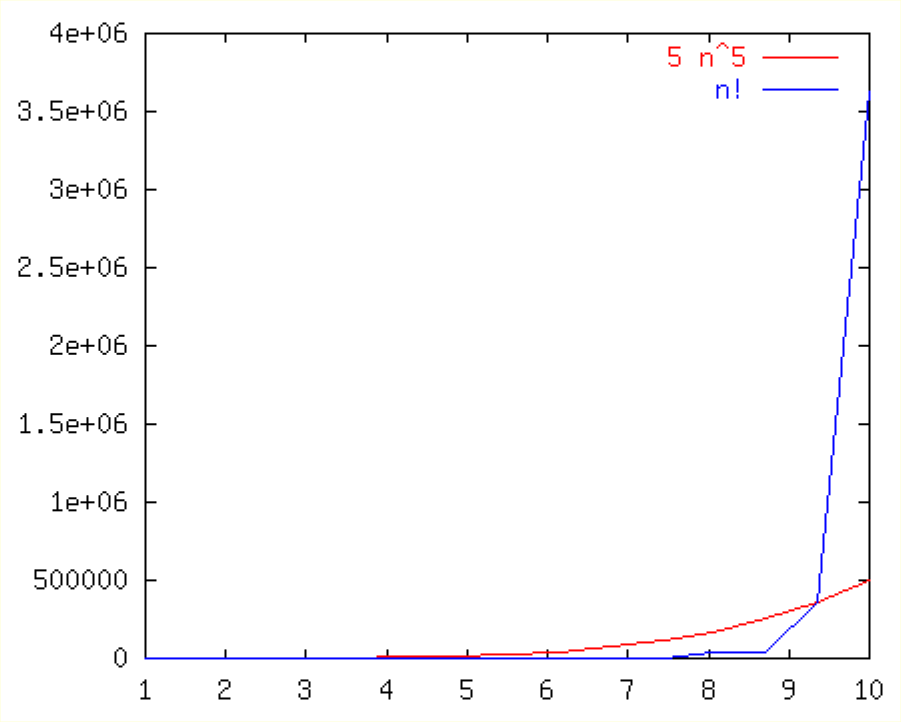




Race IV

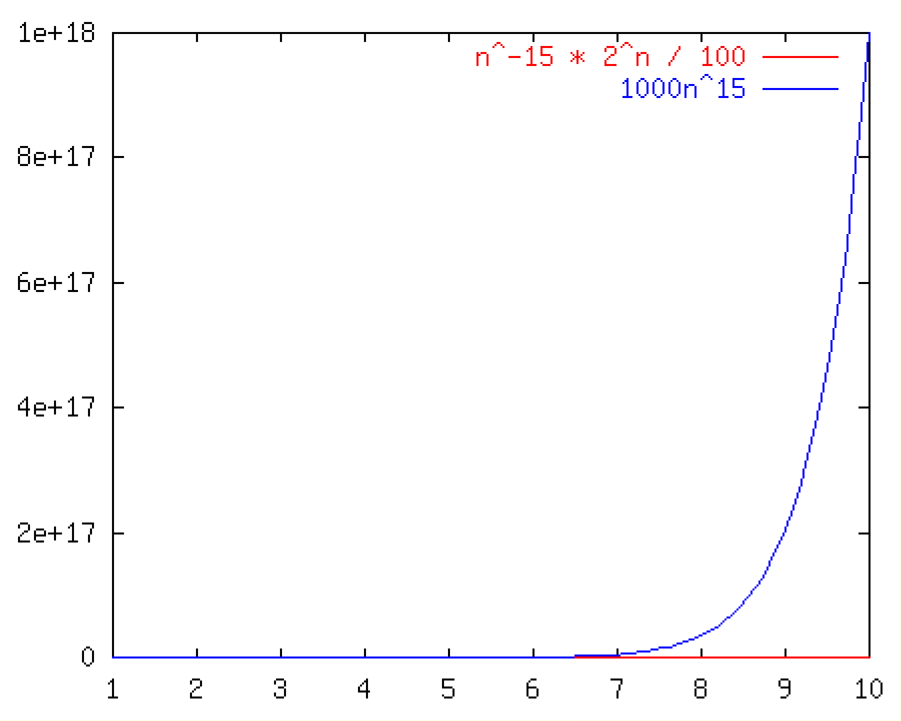
$5n^5$ vs. $n!$

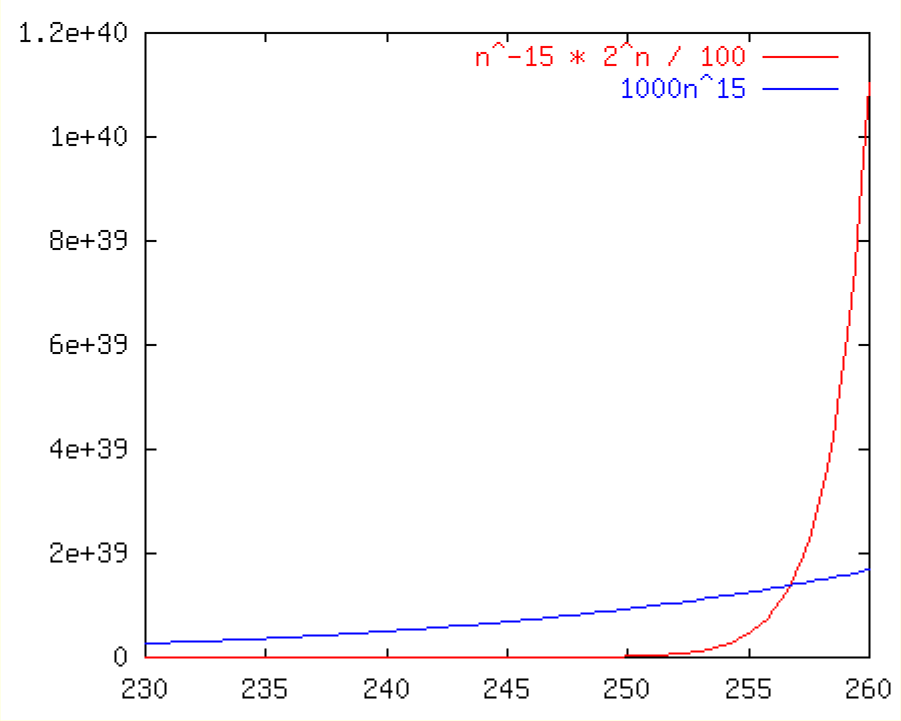




Race V

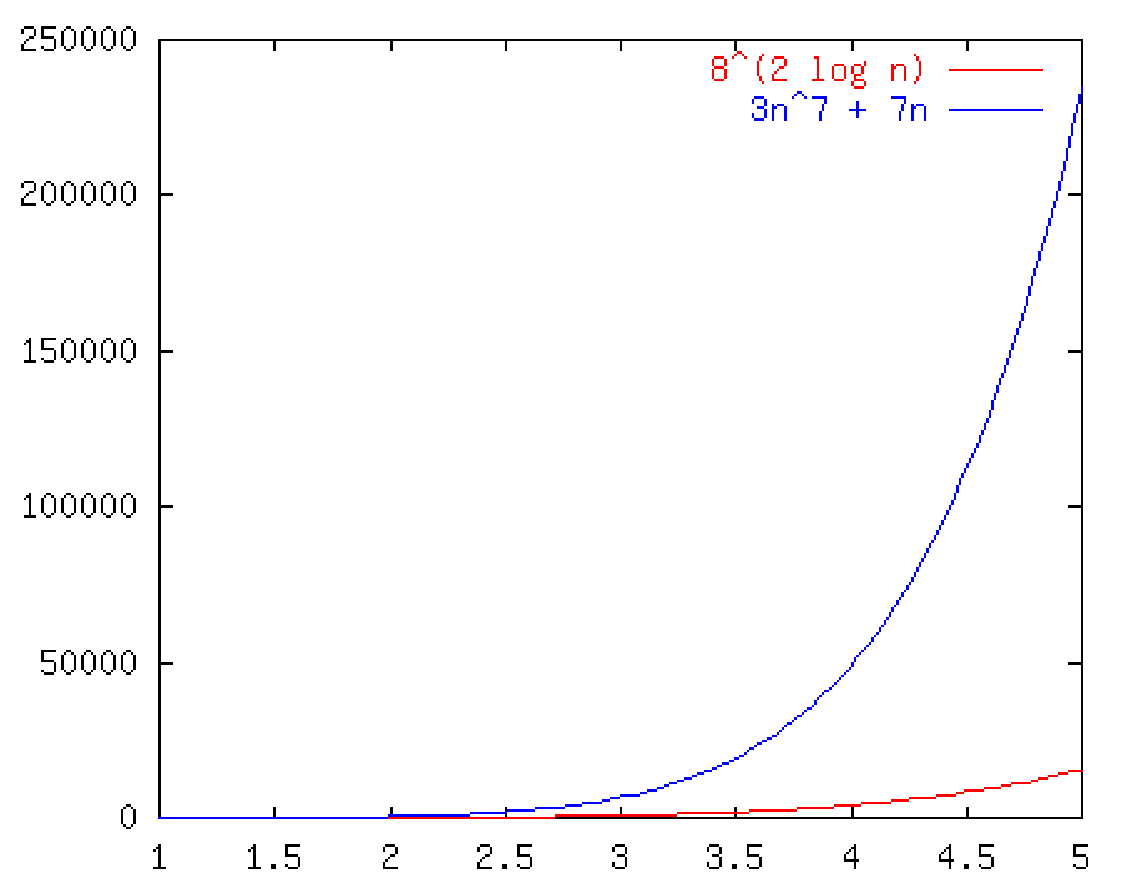
$n^{-15}2^n/100$ vs. $1000n^{15}$





Race VI

$8^{2\log(n)}$ vs. $3n^7 + 7n$



The Losers Win

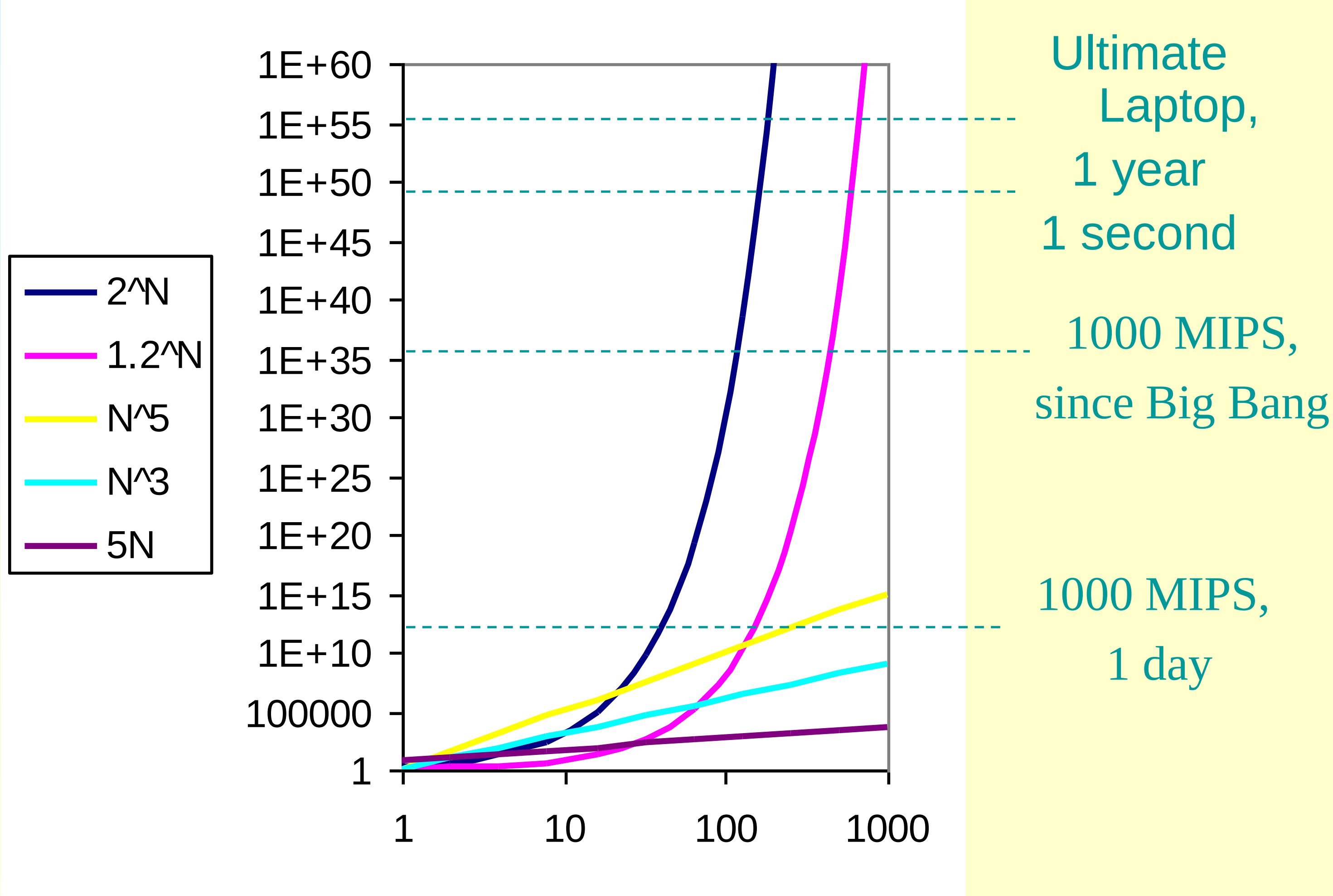
Function #1	Function #2	Better algorithm!
$n^3 + 2n^2$	$100n^2 + 1000$	$O(n^2)$
$n^{0.1}$	$\log n$	$O(\log n)$
$n + 100n^{0.1}$	$2n + 10 \log n$	**TIE** $O(n)$
$5n^5$	$n!$	$O(n^5)$
$n^{-15}2^n/100$	$1000n^{15}$	$O(n^{15})$
$8^{2\log n}$	$3n^7 + 7n$	$O(n^6)$

Common Names

constant:	$O(1)$	
logarithmic:	$O(\log n)$	
linear:	$O(n)$	
log-linear:	$O(n \log n)$	
quadratic:	$O(n^2)$	
polynomial:	$O(n^k)$	(k is a constant)
exponential:	$O(c^n)$	(c is a constant > 1)

Practical Complexity

How the various functions grow with n?



Performance Measurement

Performance measurement **is concerned with obtaining the actual space and time requirements of a program.**

To time a short event it is necessary to repeat it several times and divide the total time for the event by the number of repetitions.

- 加法规则　　　　　//并列程序段

$$T(n, m) = T_1(n) + T_2(m)$$
$$= O(\max(f(n), g(m)))$$

```
x = 0;  y = 0;
```

$T_1(n) = O(1)$

```
for ( int k = 0; k < n; k ++ )
    x ++;
```

$T_2(n) = O(n)$

```
for ( int i = 0; i < n; i++ )
    for ( int j = 0; j < n; j++ )
        y ++;
```

$T_3(n) = O(n^2)$

$$T(n) = T_1(n)+T_2(n)+T_3(n) = O(\ \max(\ 1,\ n,\ n^2\ )\ ) = O(n^2)$$

- 乘法规则　　　　//嵌套程序段

$$T(n, m) = T_1(n) * T_2(m)$$

$$= O(f(n)*g(m))$$

```
void  bubbleSort (int a[ ],  int n )
{ //对表 a[ ] 逐趟比较, n 是表当前长度
      for (int i = 1; i <= n-1; i++)
      {     //n-1趟
            for (int j = n-1; j >= i; j--)   //n-i次比较
                if (a[j-1] > a[j])
                { int tmp = a[j-1];
                   a[j-1] = a[j];
                   a[j] = tmp;
                }      //一趟比较
      }
}
```

$\mathrm{O}(f(n)*g(n)) = \mathrm{O}(n^2)$

$$\Theta \sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$$

BubblrSort

外层循环 n-1 趟

内层循环 n-i 次比较

Exercise

```
int prime(int n )
{
  int i=2, x= (int) sqrt(n);
  while ( i<=x )
   {   if ( n%i==0 ) break;
       i++;
   }
   if ( i>x ) return 1;
   else  return 0;
 }
```

$T(n) = O(\sqrt{n})$

```
int fun( int n)
{
   int i=1, s=1;
   while( s<n )
         s += ++i;
    return i;
 }
```

$T(n) = O(\sqrt{n})$

```
int sum1(int n)
{
   int p=1, s=0;
   for ( int i=1; i<=n; i++ )
    {
      p*=i;
      s+=p;
     }
   return s;
  }
```

$T(n) = O(\ n\ )$

```
int sum2 (int n)
{
  int s=0;
  for ( int i=1; i<=n; i++ )
    {
      int p=1;
      for( int j=1; j<=i; j++ )
          p*=j;
       s+=p;
     }
     return s;
 }
```

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