DATA STRUCTURES AND ALGORITHMS

Textbook:

Fundamentals of Data Structure in C++, Silicon Press

ElliS Horowitz, Sartaj Sahni, Dinesh Mehta

Teacher:

刘志昊

Email: lzh@seu.edu.cn

地 址：东南大学九龙湖校区计算机楼4楼/5楼

Evaluation:

Course Attendance: 10%,

Exercises and Projects: 20%,

Final Examination (Textbook and Course Notes allowed): 70%

References:

1 金远平，数据结构(C++描述), 清华大学出版社, 2005

2 T. A. Standish, Data Structures, Algorithms & Software Principles in C, Addison-Wesley Publishing Company, 1994

3 殷人昆，数据结构(用面向对象方法与C++语言描述)第二版，清华大学出版社，2007

Tips

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

Prerequisites:

Programming Language: C, C++

Pointer in C & C++

物有本末，事有终始。
知所先后，则近道矣。

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.

For example: 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;
}
```

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;
}
```

Basic Concepts

Purpose:

Providing the tools and techniques necessary to design and implement large-scale software systems. Discussing necessary methodologies:

- **Data abstraction and encapsulation**
- **Algorithm specification and design**
- **Performance analysis and measurement**
- **and**
- **Recursive programming**

1.1 Overview: System Life Cycle

Good programmers regard large-scare computer programs a system.

(1) Requirements

(2) Analysis

(3) Design

(4) Refinement and coding

(5) Verification and maintenance

Overview: System Life Cycle

(1) Requirements

All large programming projects begin with a set of specifications that define the purpose of the project.

input
output

(2) Analysis

begin to break the problem into manageable pieces

bottom-up
top-down

Overview: System Life Cycle

(3) Design

Approaches the system from the designer's angle of the

data objects

operations on them

TO DO

abstract data type

algorithm specification and design strategies

Example: scheduling system of university

data objects: student, courses, teachers…

operations: inserting, removing, searching…

(4) Refinement and coding

representations for data object

algorithms for operations

(5) Verification and maintenance

correctness proofs

testing with a variety of input data

removing errors

1.3 Data Abstraction and Encapsulation

(p.7)Data Encapsulation or information Hiding is the concealing of the implementation details of a data object from the outside world.

(p.7)Data Abstraction is the separation between the *specification* of a data object and its *implementation*.

DVD player example.

(p.8)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.

(p.8)An Abstract Data Type (ADT) is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is 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

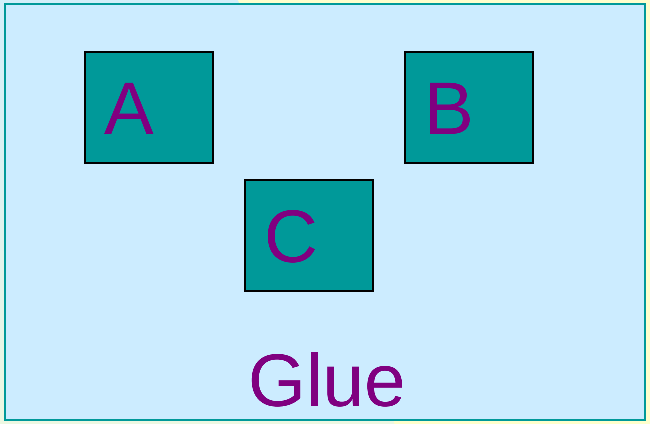
Applicaton : **data types A, B, C & Code Glue**

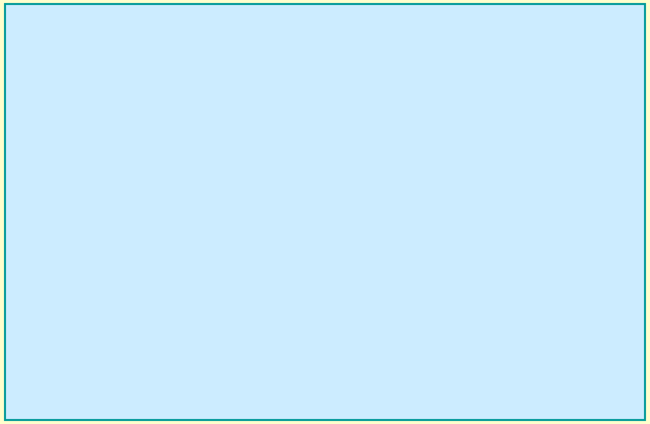
(a) a team of 4 programmers

(b) a single programmer

(2) Simplifying testing and debugging

Code with data abstraction **Code without data abstraction**





Unshaded areas represent code to be searched for bugs.

(3) Reusability

Data abstraction and encapsulation typically give rise to data structures that are implemented as distinct entities of a software system. This makes it easier to extract the code for a data structure and its operations.

(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.

1.5 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

Recursive Algorithms

Direct recursion
Indirect recursion

Similar to mathematical induction

Exercises: P32-2， P33-14

1.7 Performance Analysis and Measurement

(p.38)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

Example 1.10

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.

In P41-43 of the textbook, there is an detailed assignment of step counts to statements in C++.

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?

2 ways:

(1) introducing a global new variable count

(2) building a table

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]);
     }
}
```

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

$t_{Rsum}(n) = 2+ t_{Rsum}(n-1)$

$= 2+2+ t_{Rsum}(n-2)$

.

.

.

$= 2n+ t_{Rsum}(0)$

$=2n+2$

Example 1.14 Fibonnaci numbers

```
void  Fibonnaci (int n)
{   // compute the Fibonnaci number Fn
    if (n <=1) cout << n<< endl; { // F0=0 and F1 =1
     else { // compute Fn
        int fn; int fnm2=0; int fnm1=1;
        for (int i=2; i<=n; i++)
        {
            fn=fnm1+fnm2;
            fnm2=fnm1;
            fnm1=fn;
        } //end of for
         cout <<fn<<endl;
     }  //end of else
}
```

Let us use a table to count its total steps.

Line	s/e	frequency	total steps
1	0	1	0
2	0	1	0
3	1 (n >1)	1	1
4	0	1	0
5	2	1	2
6	1	n	n
7	0	n-1	0
8	1	n-1	n-1
9	1	n-1	n-1

10	1	n-1	n-1
11	0	n-1	0
12	1	1	1
13	0	1	0
14	0	1	0

So

for n>1, $t_{Fibonnci}(n)=4n+1$**,**

for n=0 or 1, $t_{fibonnci}(n) = 2$

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

For many programs, the time complexity is not dependent solely on the number of inputs or outputs or some other easily specified characteristic

e.g., *BinarySearch*.

Hence:

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

Asymptotic Notation (O)

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 $f(n) \le c\, g(n)$ for all n, $n>n_0$.

Example 1.15: $3n+2=O(n)$, $6*2^n+n^2=O(2^n)$,…

Note g(n) is an upper bound. It does not say anything about how good this bound is.

$n=O(n^2)$, $n=O(2^n)$, …,

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.

Asymptotic Notation (Ω)

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

Example 1.16: $3n+2=\Omega(n)$, $6*2^n+n^2=\Omega(2^n)$,…

Note g(n) is an lower bound.

$n^2=O(n\ )$, $2^n= O(n)$, …,

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

as large as possible.

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

Theory 1.3: if $f(n)=a_m n^m+\ldots+ a_1 n+a_0$, and $a_m>0$ then $f(n)= \Omega(n^m)$.

Asymptotic Notation (Θ)

Definition [Θ]: $f(n)=\Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_0 such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all n, $n>n_0$.

Example 1.17: $3n+2=\Theta(n)$, $6*2^n+n^2=\Theta(2^n)$,…

The theta notation is more precise than both the "big oh" and omega notations.

$f(n)=\Theta(g(n))$ iff g(n) is both an upper and lower bound on f(n).

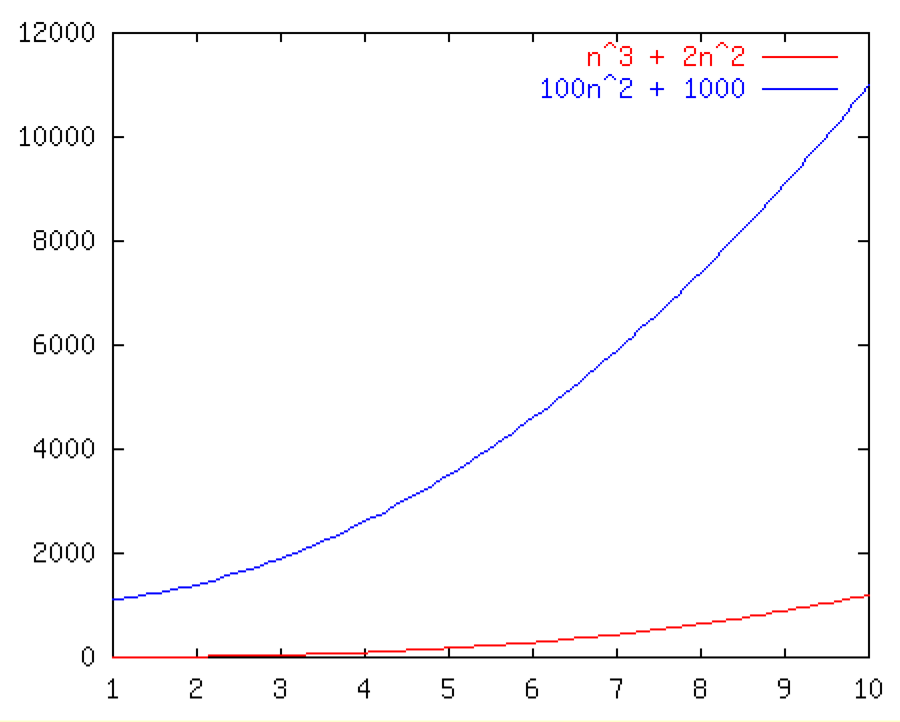
Theory 1.4: if $f(n)=a_m n^m+\ldots+a_1 n+a_0$, and $a_m>0$ then $f(n)=\Theta(n^m)$.

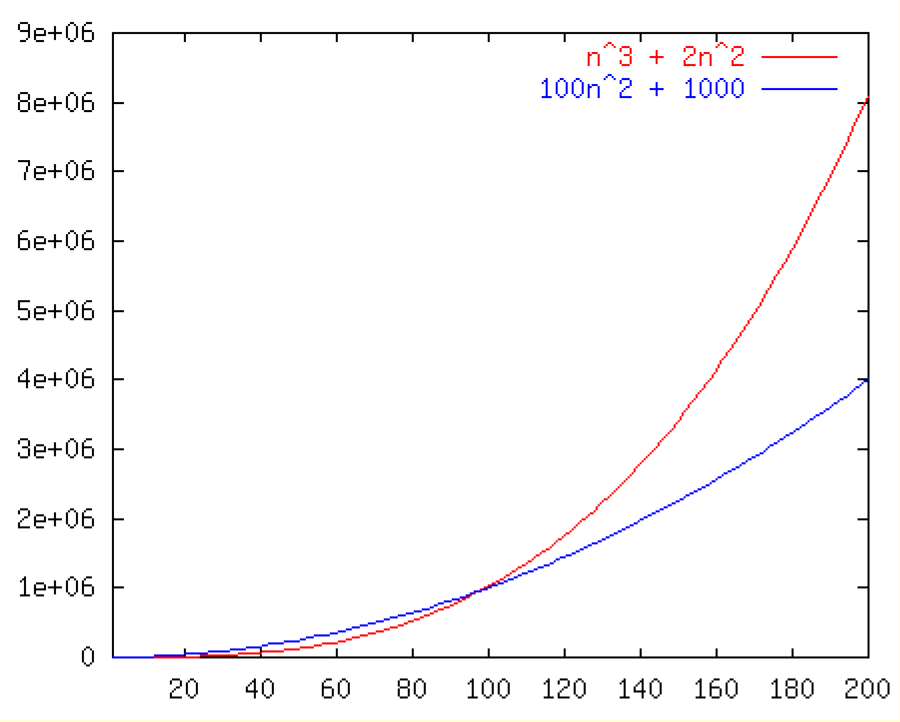
A Few Comparisons

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

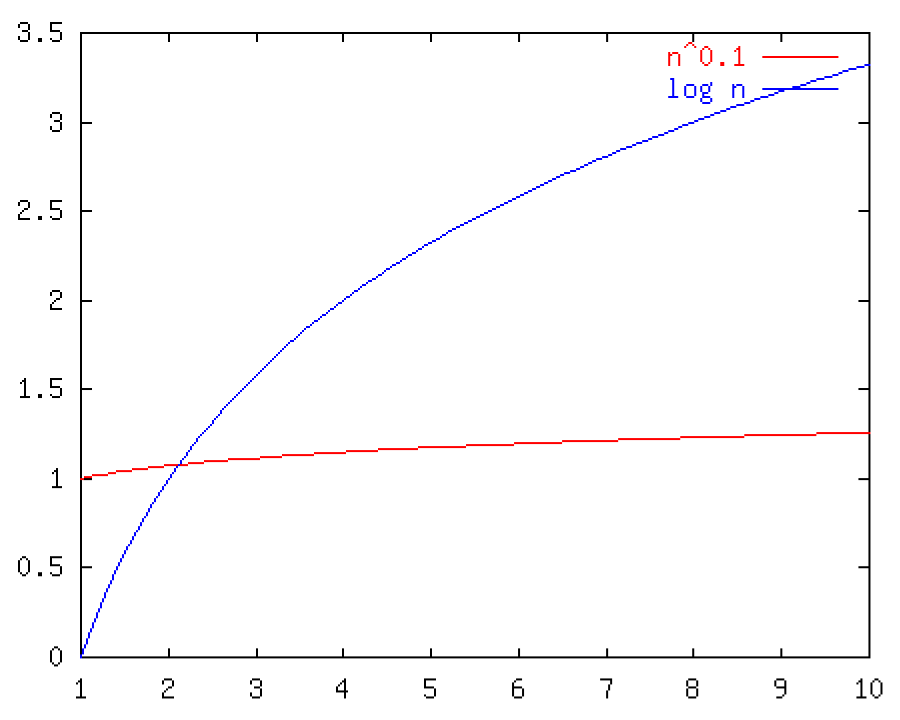
$\mathbf{n^3 + 2n^2}$ vs. $\mathbf{100n^2 + 1000}$

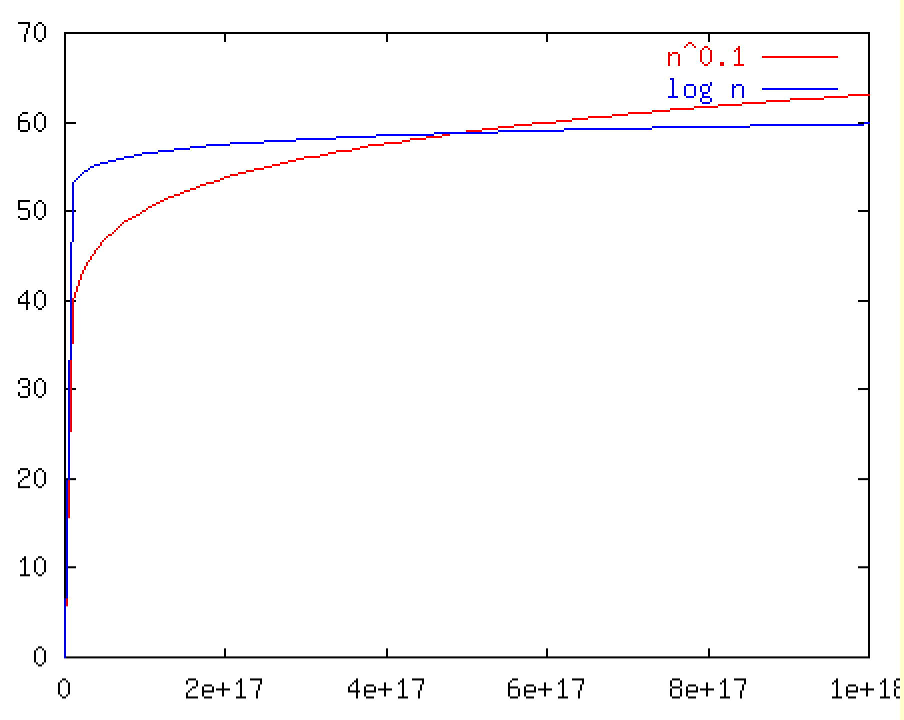




Race II

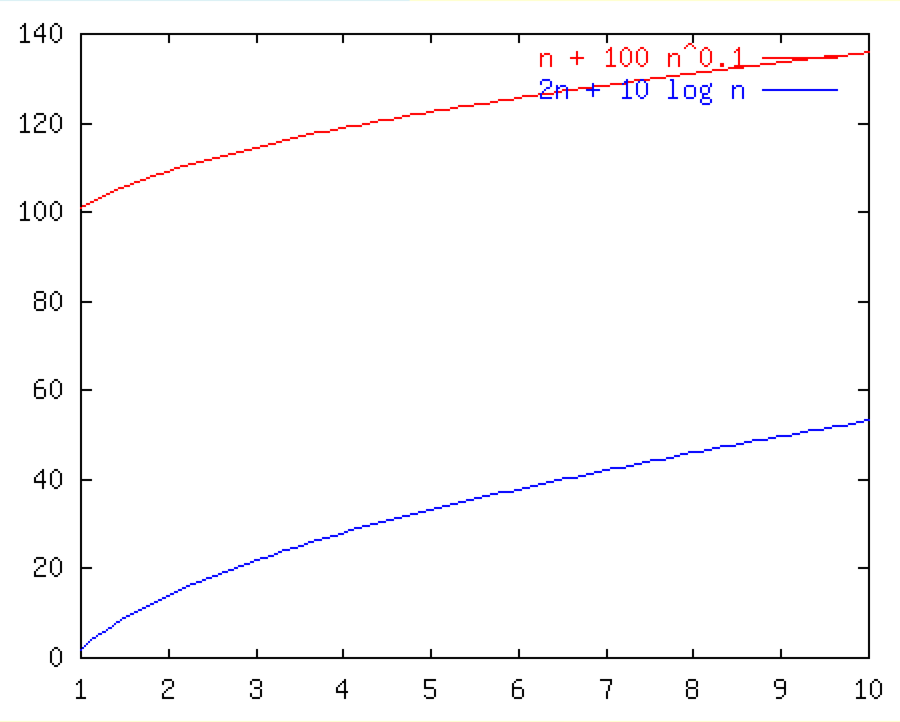
$n^{0.1}$ vs. $\log n$

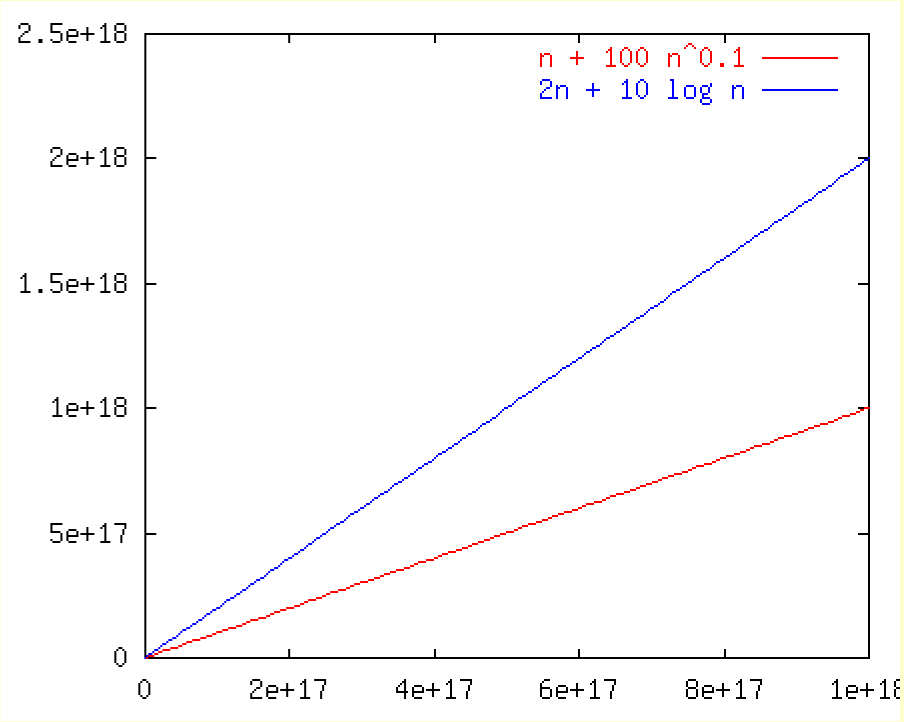




Race III

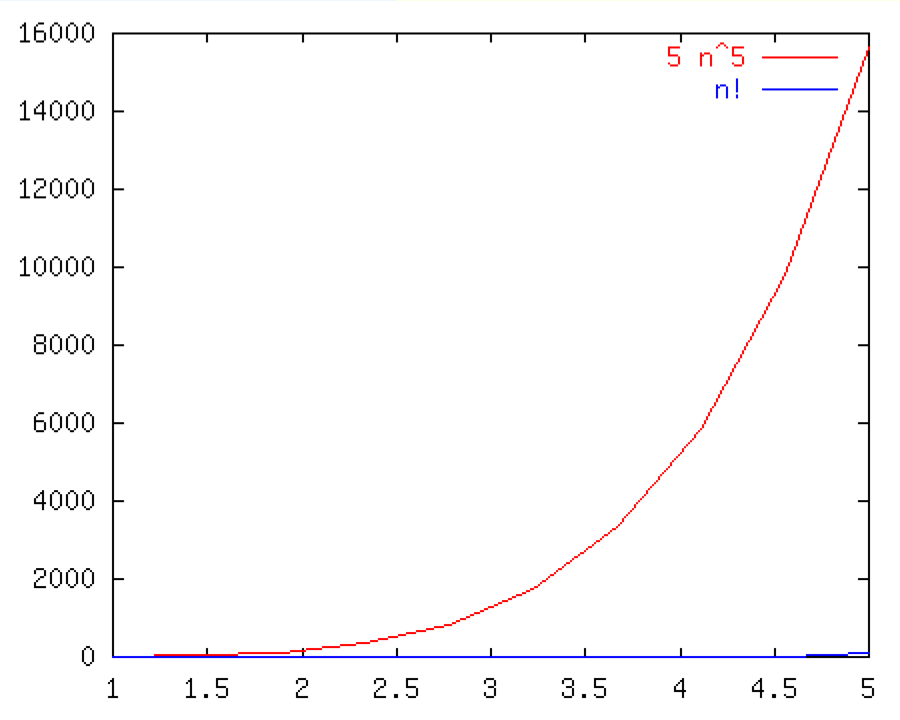
$n + 100n^{0.1}$ vs. $2n + 10 \log n$

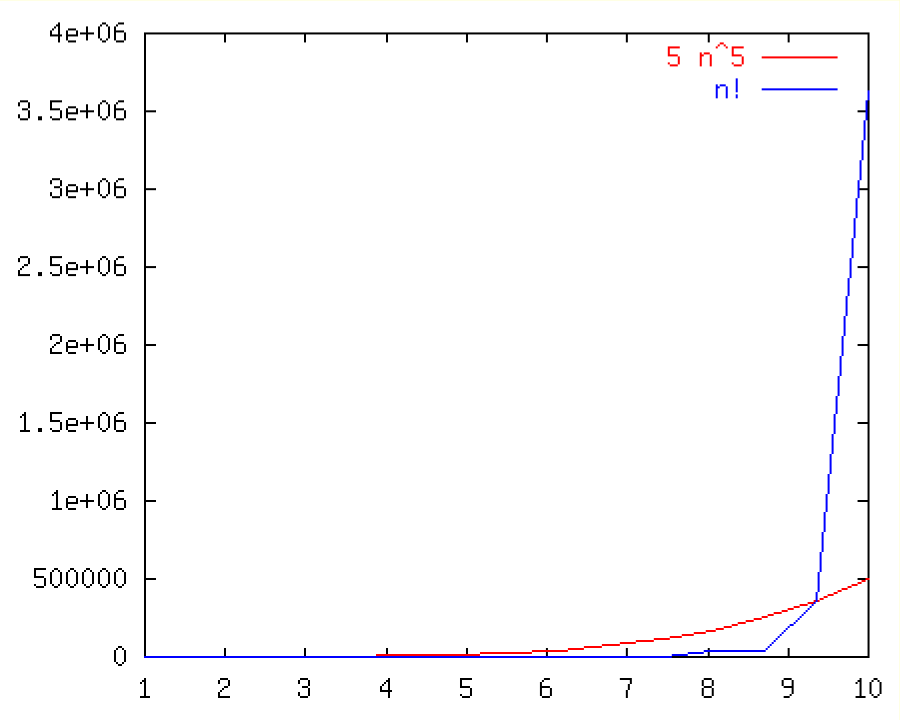




Race IV

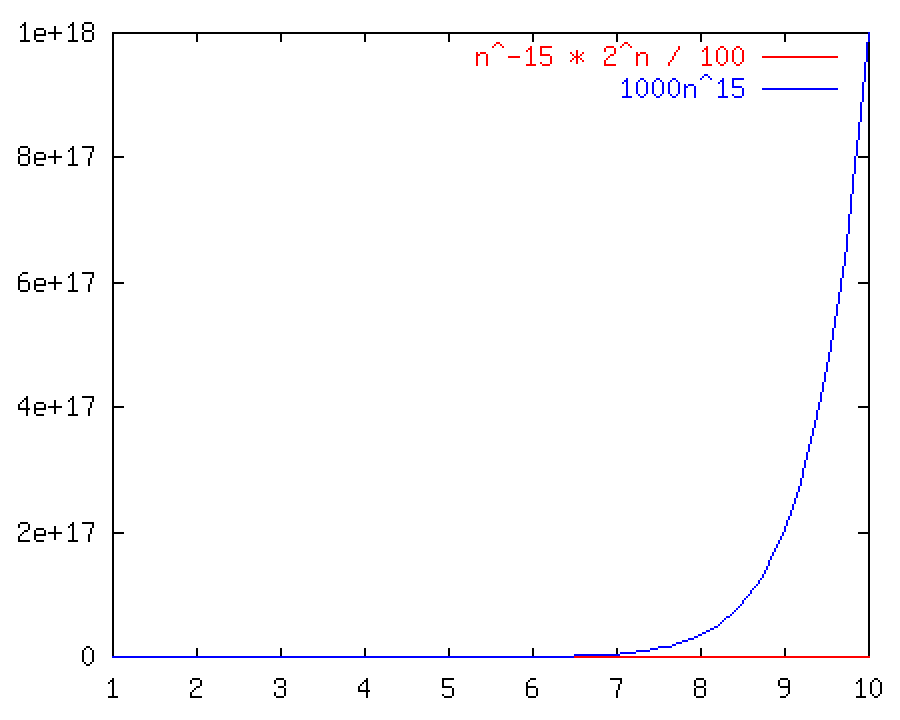
$5n^5$ vs. **n!**

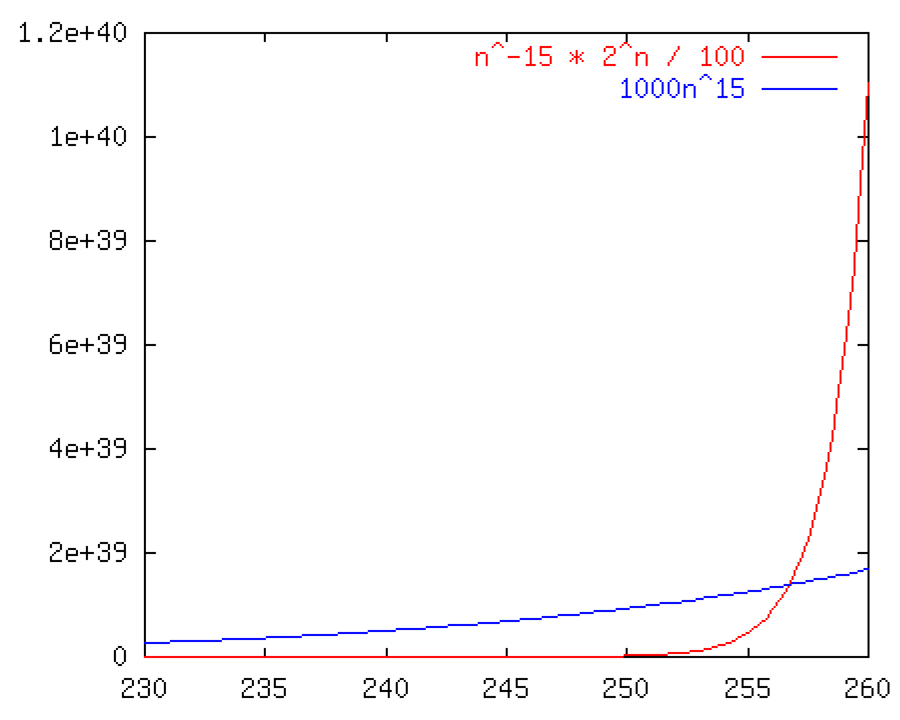




Race V

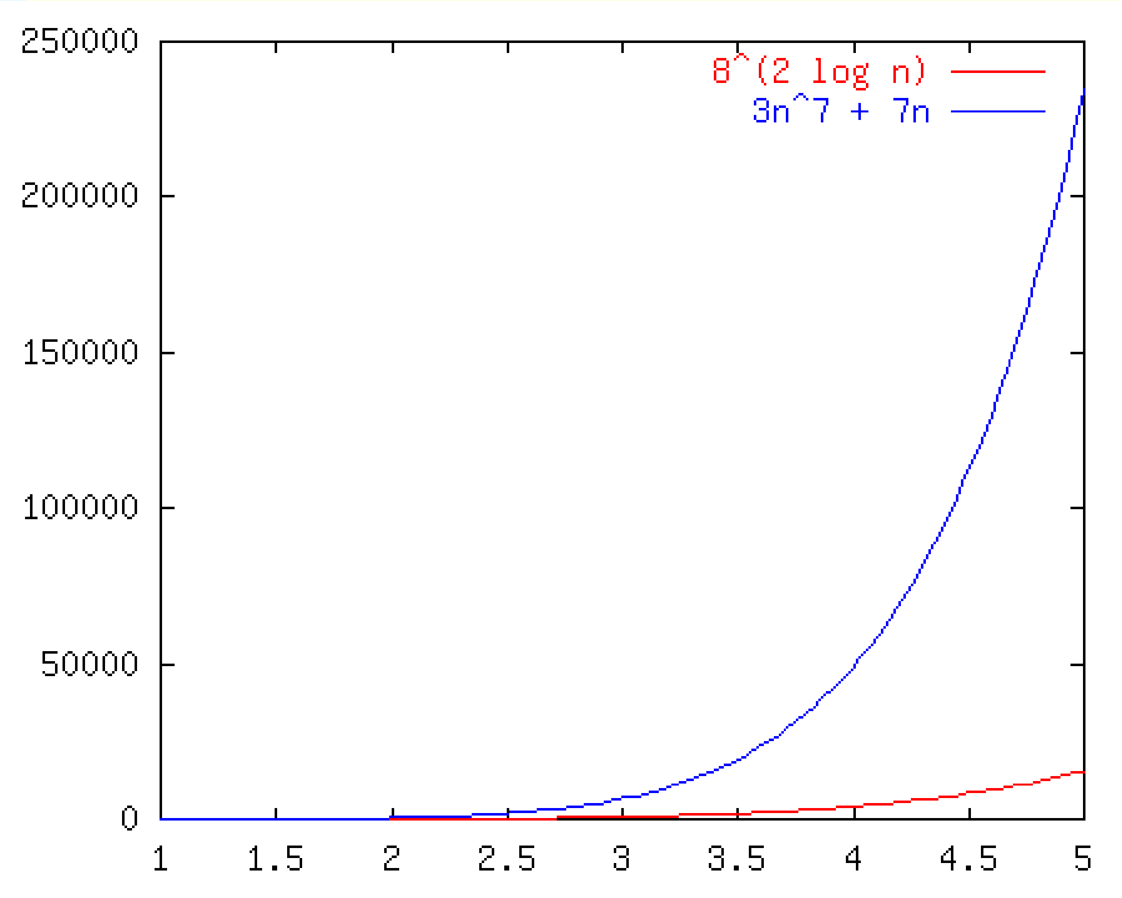
$n^{-15}2^n/100$ vs. $1000n^{15}$





Race VI

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



The Losers Win

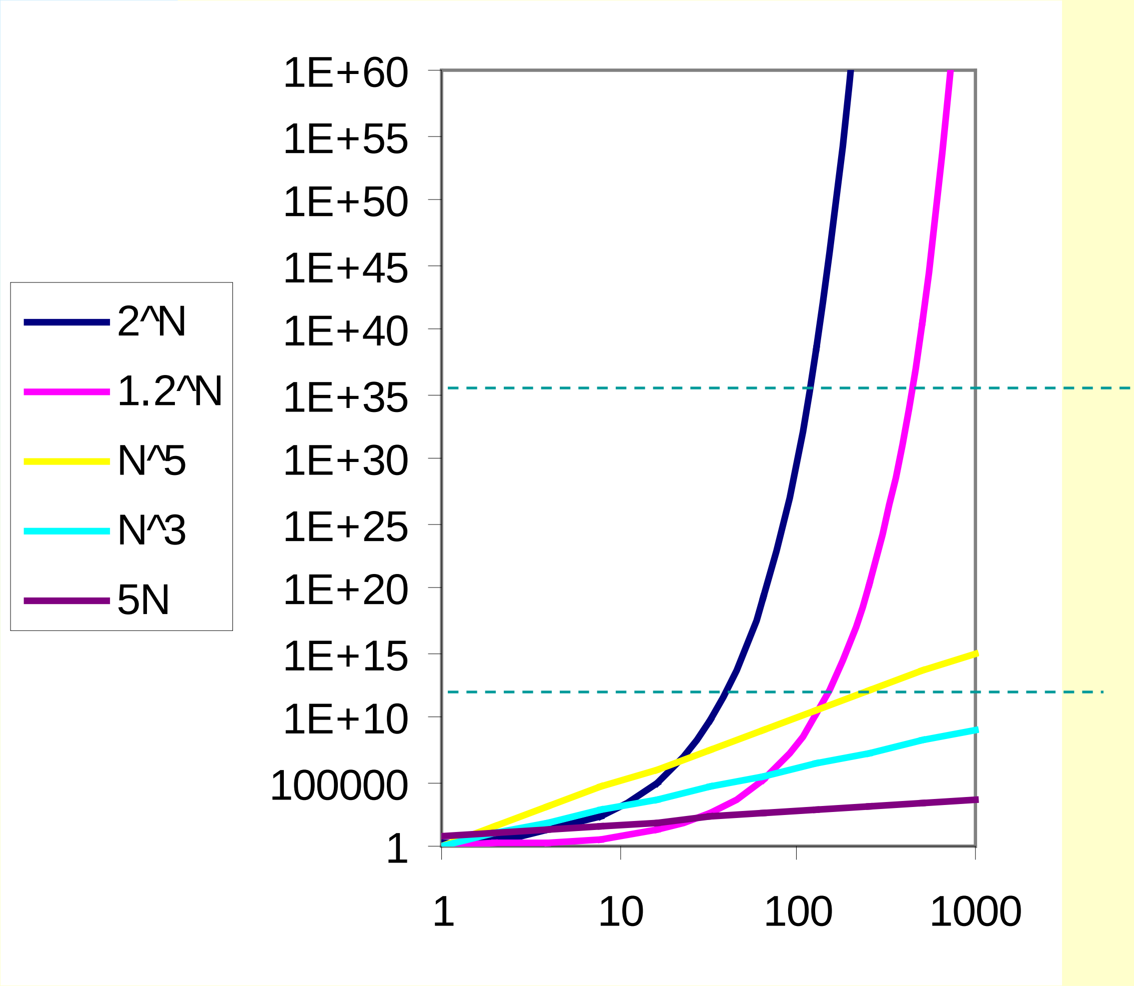
Function #1	Function #2	Better algorithm!
$n^3 + 2n^2$	$100n^2 + 1000$	$O(n^2)$
$n^{0.1}$	$\log n$	$O(\log n)$
$n + 100n^{0.1}$	$2n + 10 \log n$	**TIE** $O(n)$
$5n^5$	$n!$	$O(n^5)$
$n^{-15}2^n/100$	$1000n^{15}$	$O(n^{15})$
$8^{2\log n}$	$3n^7 + 7n$	$O(n^6)$

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?



1000 MIPS,
since Big Bang

1000 MIPS,
1 day

n	f(n)=n	f(n)=$n\log_2 n$	f(n)=n^2	f(n)=n^4	f(n)=n^{10}	f(n)=2^n
10	.01μs	.03 μs	.1 μs	10 μs	10s	1 μs
20	.02 μs	.09 μs	.4 μs	160 μs	2.84h	1 ms
30	.03 μs	.15 μs	.9 μs	810 μs	6.83d	1 s
40	.04 μs	.21 μs	1.6 μs	2.56ms	121d	18m
50	.05 μs	.28 μs	2.5 μs	6.25ms	3.1y	13 d
100	.1 μs	.66 μs	10 μs	100 ms	3171y	$4*10^{13}$y
10^3	1 μs	9.66 μs	1ms	16.67m		
10^4	10 μs	130 μs	100ms	115.7d		
10^5	100 μs	1.66ms	10s	3171y		

Table 1.8: Times on a 1-billion-steps-per-second computer

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.

Let us look at the following program:

```
int  SequentialSearch (int *a, const int n, const int x )
{ // Search a[0:n-1].
    int  i;
    for (i=0; i < n && a[i] != x; i++;)
        if (i == n) return -1;
    else return i;
}
```

```
void TimeSearch ( )
{
   int a[1000], n[20];
   const long r[20] = {300000, 300000, 200000, 200000,
   100000, 100000, 100000, 80000, 80000, 50000, 50000,
    25000, 15000, 15000, 10000, 7500, 7000, 6000, 5000,
    5000 };

   for ( int j=0; j<1000; j++ ) a[j] = j+1; //initialize a
   for ( j=0; j<10; j++ ) {  //values of n
       n[j] = 10*j;  n[j+10] = 100*( j+1 );
    }

   cout << " n        total        runTime" << endl;
```

```
for ( j=0; j<20; j++ ) {
    long start, stop;
    time (&start);                                  // start timer
    for ( long b=1; b<=r[j]; b++ )
            int k = seqsearch(a, n[j], 0 );  //unsuccessful search
    time (&stop);                                   // stop timer
    long totalTime = stop - start;
    float runTime =  (float) (totalTime) / (float)(r[j]);
    cout << " " << n[j] << " " << totalTime << " " << runTime
             << endl;
  }
}
```

The results of running *TimeSearch* are as in the next slide.

n	total	runTime	n	total	runTime
0	241	0.0008	100	527	0.0105
10	533	0.0018	200	505	0.0202
20	582	0.0029	300	451	0.0301
30	736	0.0037	400	593	0.0395
40	467	0.0047	500	494	0.0494
50	565	0.0056	600	439	0.0585
60	659	0.0066	700	484	0.0691
70	604	0.0075	800	467	0.0778
80	681	0.0085	900	434	0.0868
90	472	0.0094	1000	484	0.0968

Times in hundredths of a second, the plot of the data can be found in Fig. 1.7.

Issues to be addressed:

(1) Accuracy of the clock

(2) Repetition factor

(3) Suitable test data for worst-case or average performance

(4) Purpose: comparing or predicting?

(5) Fit a curve through points

Exercises:

P72-10