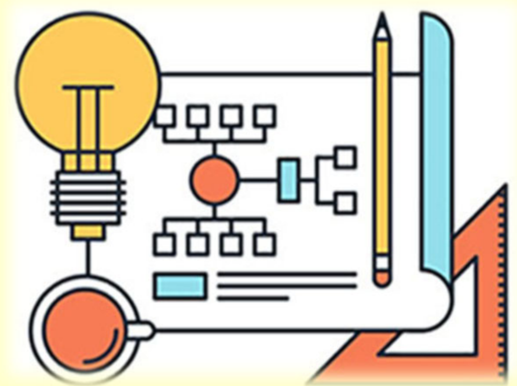
Data Structures

Wenjia Wu（吴文甲）

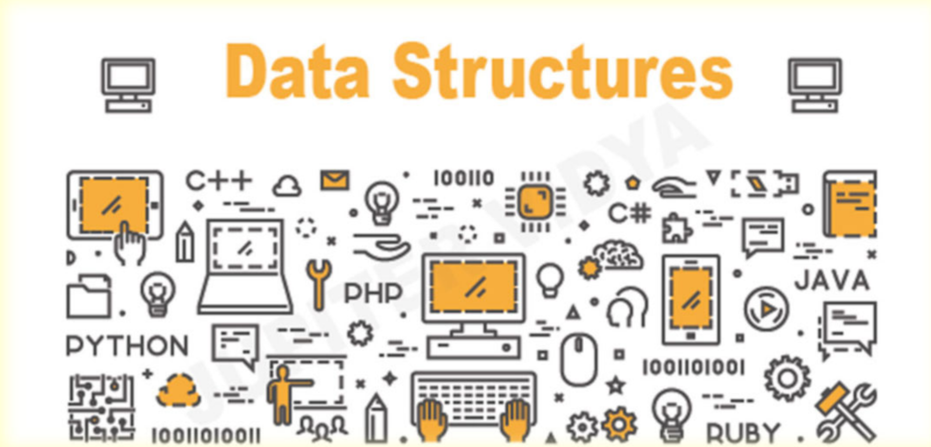
wjwu@seu.edu.cn

Southeast University

课程信息



- 课程名称：数据结构（双语）
- 课程编号：BJSL0060
- 课程性质：计算机大类学科基础课程
- 学分/学时：4 学分 / 80 学时 （授课 64 + 实验 16）
- 考核方式： 考试



- 主讲教师：吴文甲，wjwu@seu.edu.cn

- **互动方式**

- 课程QQ群：786600104

- 办公室：计算机楼166室

群名称:数据结构2023
群　号:786600104

Teaching assistant： Yipeng Rong(荣逸鹏),

assignmenthub@163.com

References:

1. E. Horowitz, S. Sahni, D. Mehta, Fundamentals of Data Structures in C++, 2E, Silicon Press, 2007

2. 殷人昆, 数据结构 (用面向对象方法与C++语言描述, 第2版), 清华大学出版, 2007

3. C. A. Shaffer, Data Structures and Algorithm Analysis, 3E, 2013

4. 金远平, 数据结构 (C++描述), 清华大学出版社, 2005

Prerequisites

- C++ Programming
- Discrete Mathematics

Total Class Hours: **64**

Week 1-16

Total Lab. Hours: **16**

时间、地点安排：待定

Program Language: C++

IDE: Microsoft Visual Studio 2019

Assignments and projects

- Should be handed to the teaching assistant.

Evaluation

Assignments: 20%,

Exercises and Projects: 30%,

Final Examination (Textbook and Course Notes allowed): 50%

Tips

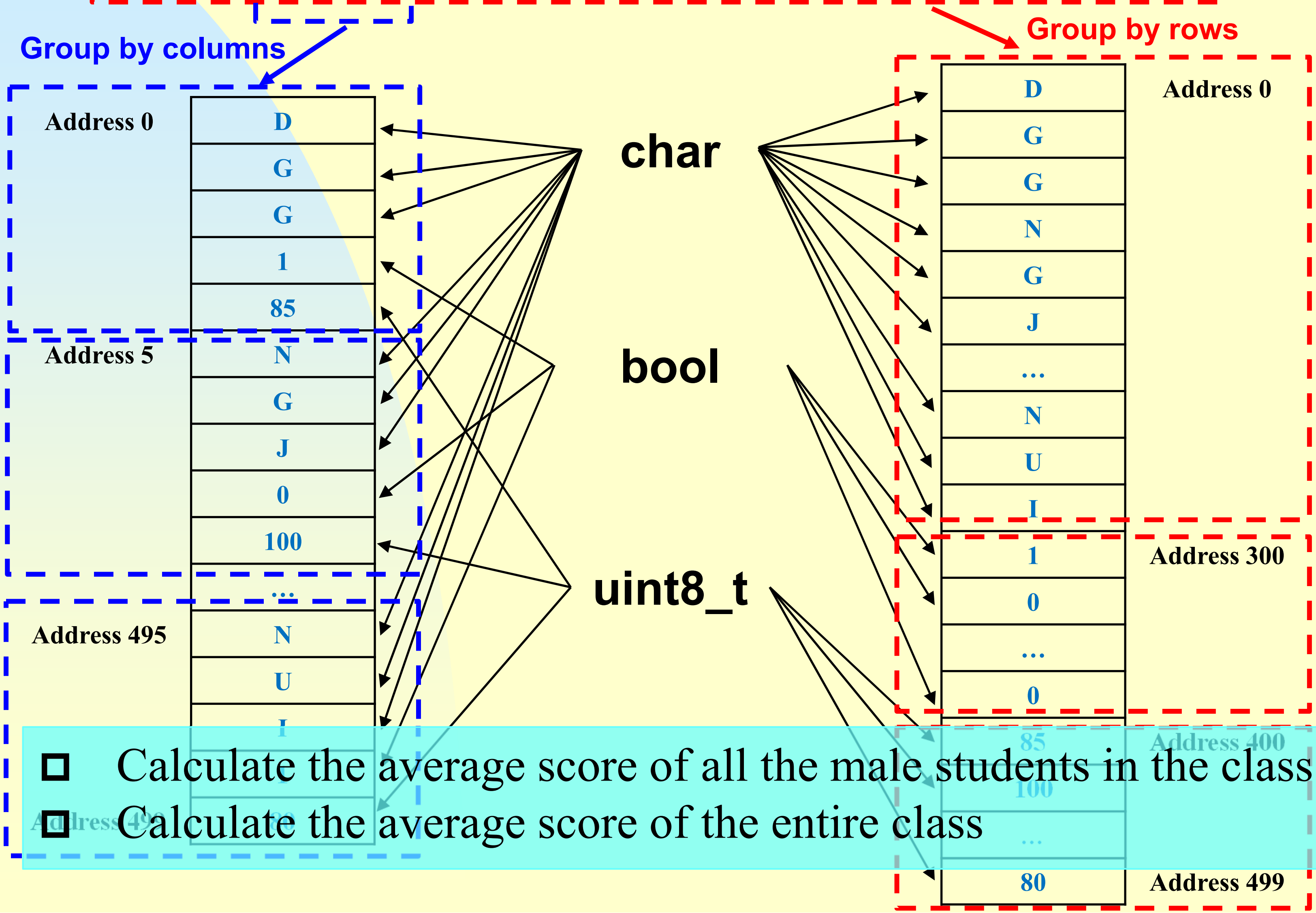
- Make good use of your time in class
  - Listening
  - Thinking
  - Taking notes
- Expend your free time
  - Go over
  - Programing
- Take a pen and some paper with you
  - Notes
  - Exercises

In computer science, a **data structure** is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

https://en.wikipedia.org/wiki/Data_structure

Name	DGG	NGJ	DSB	NCG	GHJ	…	NUI
Gender	1	0	0	0	0	…	1
Score	85	100	88	75	66	…	80

1: male
0: female



- Calculate the average score of all the male students in the class
- Calculate the average score of the entire class

Algorithms + Data Structures = Programs

--Niklaus Wirth

Data Structures (topics)

1. Basic Concepts
2. Arrays
3. Stacks and Queues
4. Linked Lists
5. Trees
6. Graphs
7. Sorting
8. Hashing
9. Efficient Binary Search Trees
10. Multiway Search Trees

Basic Concepts

Purpose:

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

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

Overview: System Life Cycle

(1) Requirements

specifications of purpose

input

output

(2) Analysis

break the problem into manageable pieces

bottom-up

top-down

Overview: System Life Cycle

(3) Design

a SYSTEM? (from the designer's angle)

data objects

operations on them

TO DO

abstract data type

algorithm specification and design

Example: scheduling system of university

??

??

(4) Refinement and coding

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

(5) Verification and maintenance

- correctness proofs
- testing
- error removal
- update

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

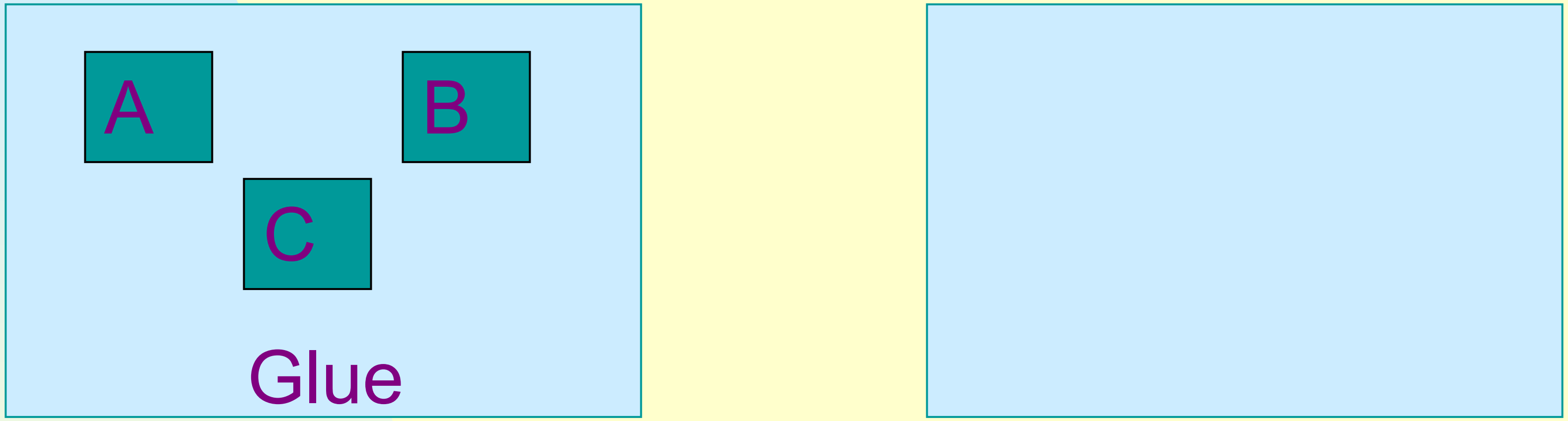
Applicaton : **data types A, B, C & Code Glue**

(a) a team of 4 programmers

(b) a single programmer

Testing and debugging

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



Unshaded areas represent code to be searched for bugs.

(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.

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

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;
}
```

Recursive Algorithms

- **Function: a set of instructions that perform a logical operation, perhaps a very complex and long operation, can be grouped together as a function.**
- **Functions call themselves (direct recursion) before they are done.**
- **Functions call other functions that again invoke the calling function (indirect recursion).**

factorial Function (*n!*)

- $$factorial(n) = \begin{cases} 1, \; n = 0,1 \\ n \times (n-1) \times \cdots \times 1, \; n \geq 2 \end{cases}$$

```
int factorial (int n){
        int p = 1;
        for (int i = 2; i  <= n; i++) {
                p *= i;
        }
        return p;
}
```

factorial Function (*n!*)

- $factorial(0) = 1$ (by definition) $= 1$
- $factorial(1) = 1 = 1 \times factorial(0)$
- $factorial(2) = 2 \times 1 = 2 \times factorial(1)$
- $factorial(3) = 3 \times 2 \times 1 = 3 \times factorial(2)$
- $factorial(4) = 4 \times 3 \times 2 \times 1 = 4 \times factorial(3)$
- $factorial(5) = 5 \times 4 \times 3 \times 2 \times 1 = 5 \times factorial(4)$
- $factorial(6) = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 6 \times factorial(5)$

factorial Function (*n!*)

- $factorial(n) = \begin{cases} 1, \; n = 0 \\ n \times factorial(n-1), \; n \geq 1 \end{cases}$

```
int factorial (int n){
        if ( n == 0)
                return 1;
        else
                return n*factorial(n-1);
}
```

Understanding recursion

- **Can you define the original problem in terms of smaller problem(s) of the same type?**

Example: $factorial(n) = n \times factorial(n\text{-}1)$ for $n > 0$

- **Does each recursive call diminish the size of the problem?**
- **As the problem size diminishes, will you eventually reach a “base case” that has an easy solution?**

Example: $factorial(0) = 1$

Recursive & Iterative

- **Anything that can be solved *iteratively* can be solved *recursively* and vice versa.**
- **Sometimes a *recursive* solution can be expressed more simply and succinctly than an *iterative* one.**

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 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

float Abc(**float** a, **float** b, **float** c)

{

        return a+b+b*c+(a+b-c)/(a+b)+4.0;

}

S_P(instance characteristics) = 0

Performance Analysis

Example

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

```
{
        float s = 0;
        for(int i = 0; i < n; i++)
                s += a[i];
        return s;
}
```

$S_{Sum}(n) = 0$

Performance Analysis

Example

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

$\boldsymbol{n}$

each call requires 4 words (n, a, return value, return address)

the depth of recursion is

$\boldsymbol{n+1}$

$$S_{Rsum}(n) =$$

$$4(n+1)$$

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.

Example:

return a + b + b*c + (a + b – c)/(a + b) + 4.0;

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

Time complexity

Detailed assignment of step counts to statements in C++:

(1) Comments

(2) Declarative statements

(3) Expressions and assignment statements

(4) Iteration statements

for(<init-stmt>;<expr1>;<expr2>)

while(<expr>) **do**

do … **while** <expr>

Time complexity

Detailed assignment of step counts to statements in C++:

(5) Switch statement

switch(<expr>){

case cond1: <statement1>

case cond2: <statement2>

…

default: <statement>

}

Time complexity

Detailed assignment of step counts to statements in C++:

(6) If-else statement

if(<expr>) <statements1>

else <statements 2>

(7) Function invocation

Time complexity

Detailed assignment of step counts to statements in C++:

(8) Memory management statements

(9) Function statements

(10) Jump statements

Our main concern:

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

2 ways:

(1) count

(2) 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]);
     }
}
```

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

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

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
}
```

Example 1.14 Fibonnaci numbers

```
1 void Fibonnaci (int n)
2 {   // compute the Fibonnaci number F_n
3    if (n <= 1) { cout << n<< endl; } // F_0 = 0 and F_1 = 1
4    else { // compute F_n
5       int fn; int fnm2=0; int fnm1=1;
6       for (int i=2; i<=n; i++)
7       {
8          fn=fnm1+fnm2;
9          fnm2=fnm1;
10         fnm1=fn;
11      } //end of for
12      cout <<fn<<endl;
13   } //end of else
14 }
```

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 $n > 1$, $t_{\text{Fibonnci}}(n) = 4n + 1$,

for $n = 0$ or 1, $t_{\text{Fibonnci}}(n) = 2$

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** $f(n) \le cg(n)$ **for all** n, $n > n_0$.

Example: $3n + 2 = O(n)$

$$6*2^n + n^2 = O(2^n)$$

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 $O(3n)$.

Theorem 1.2: If $f(n) = a_m n^m + \dots + a_1 n + a_0$, 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

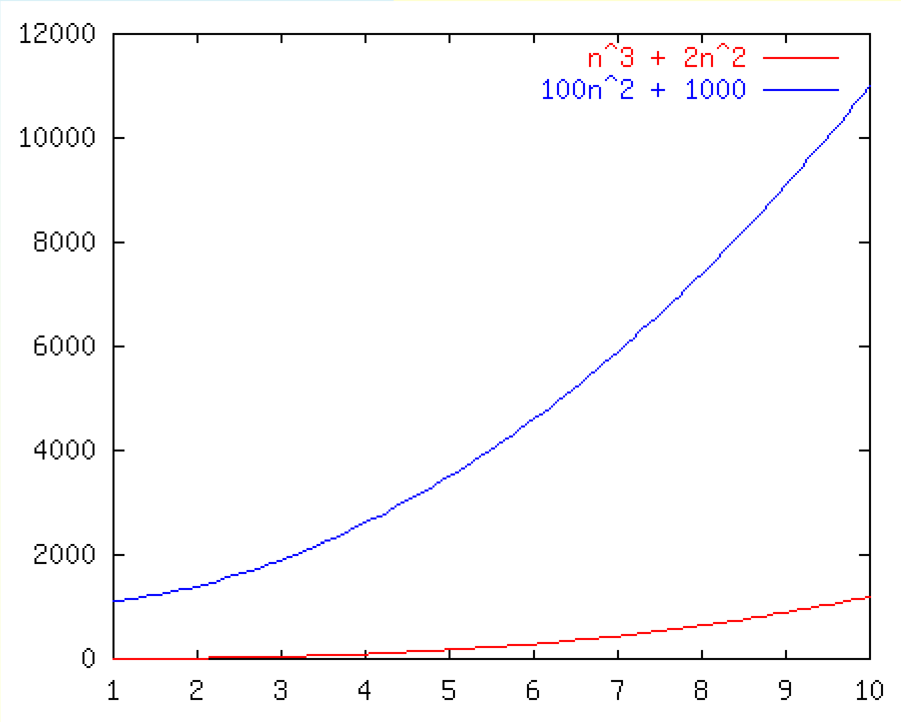
Θ --- equal bound

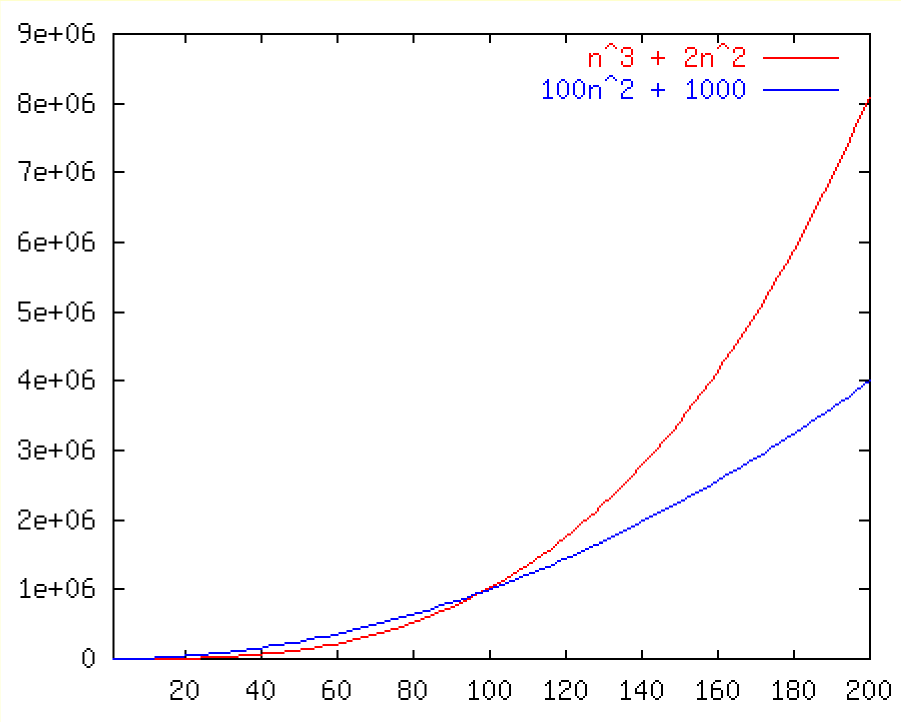
A Few Comparisons

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

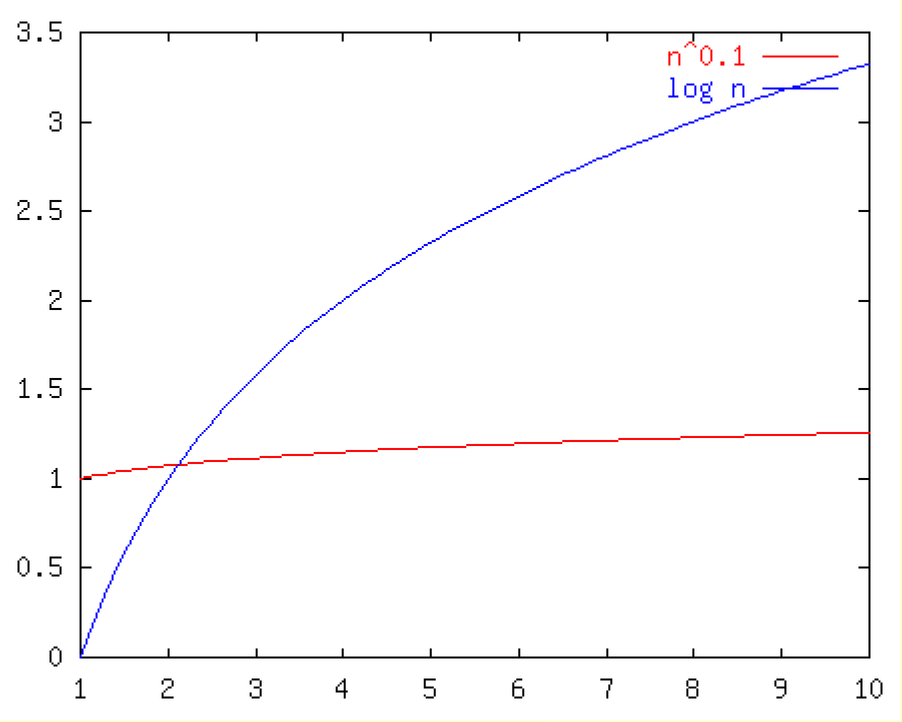
$\mathbf{n^3 + 2n^2}$ vs. $\mathbf{100n^2 + 1000}$

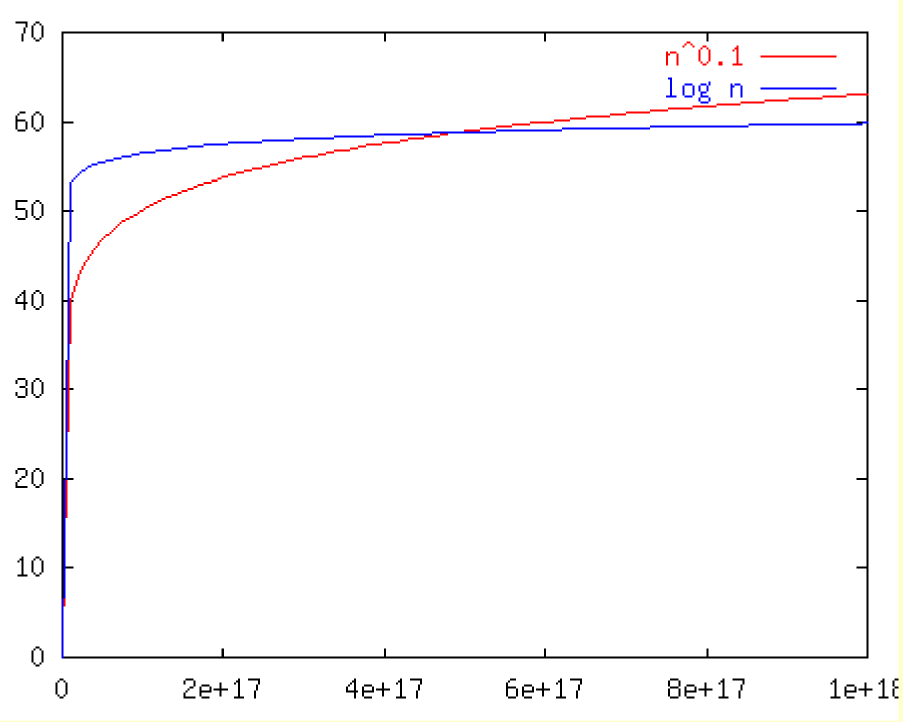




Race II

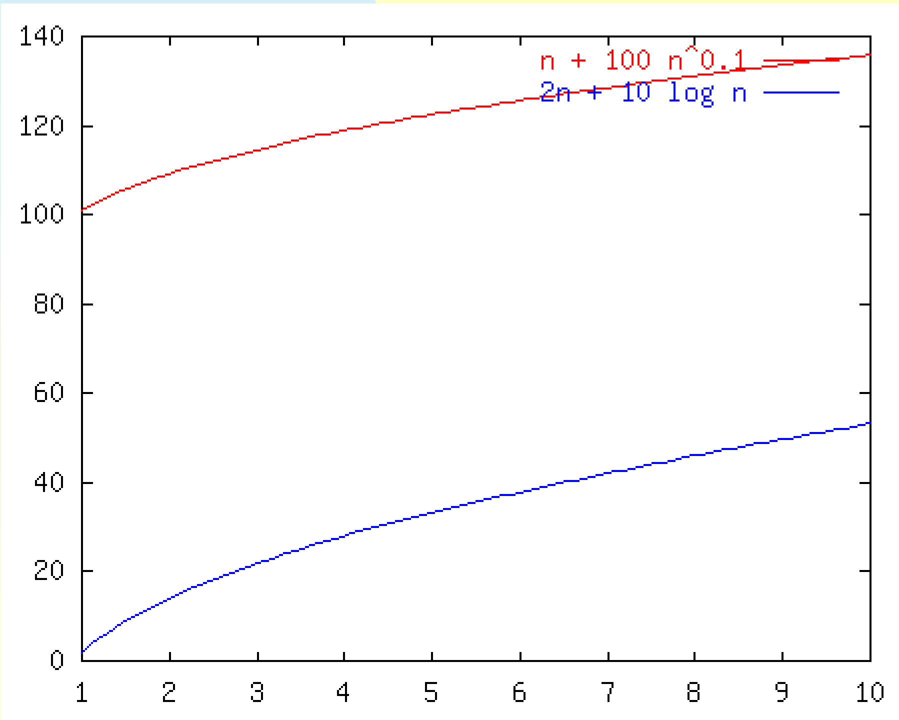
$n^{0.1}$ vs. $\log n$

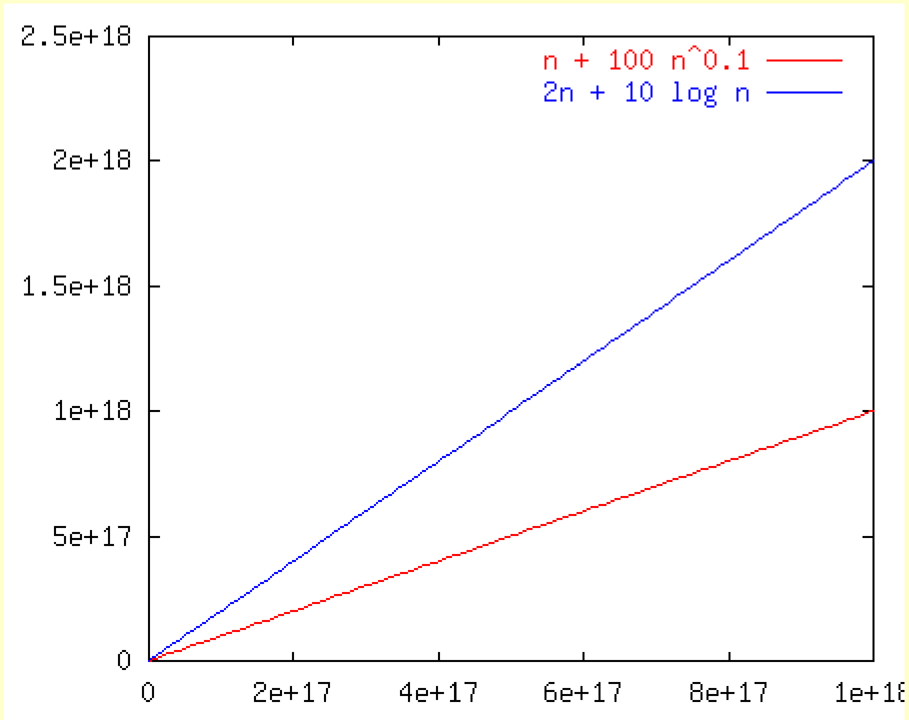




Race III

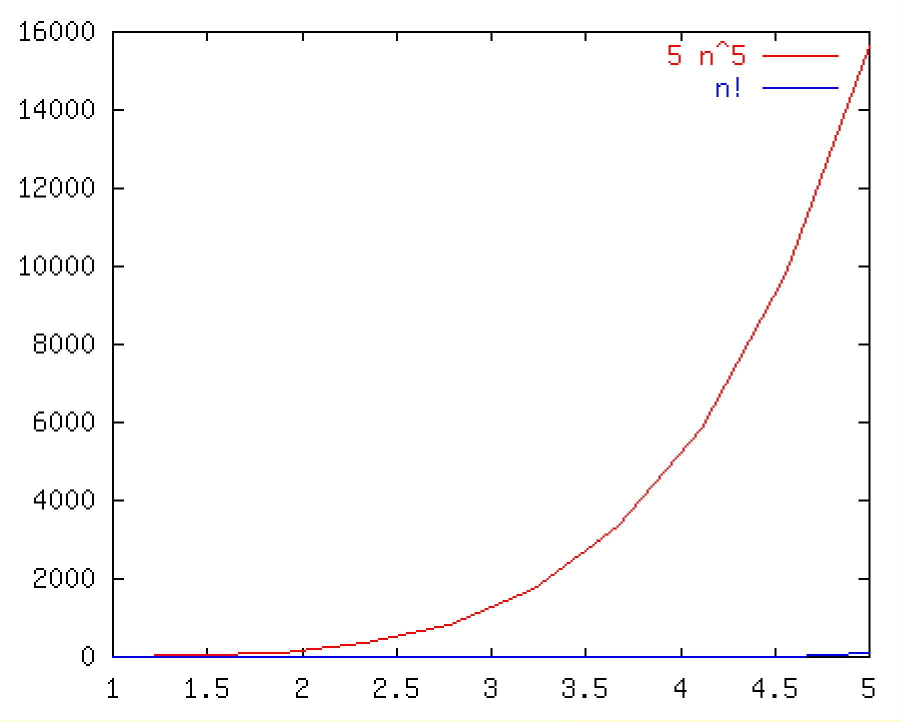
$\mathbf{n + 100n^{0.1}}$ vs. $\mathbf{2n + 10 \log n}$

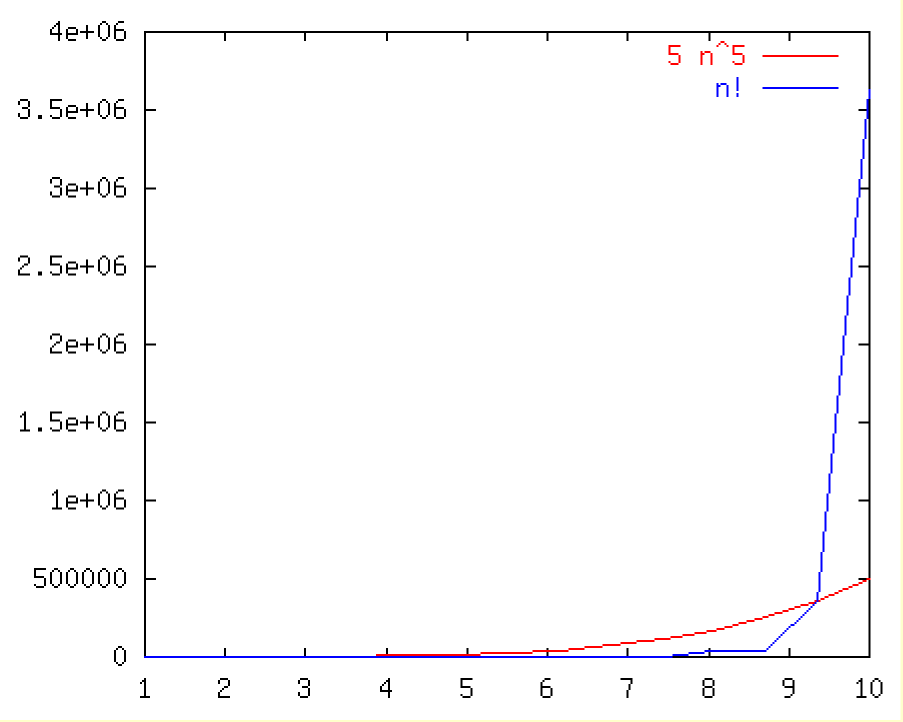




Race IV

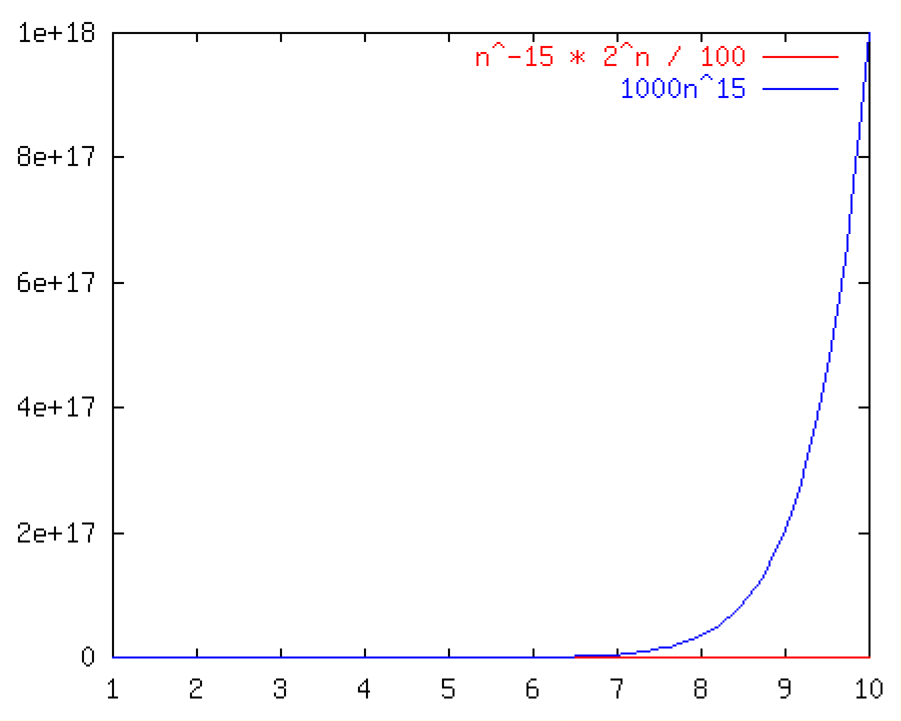
$5n^5$ vs. `n!`

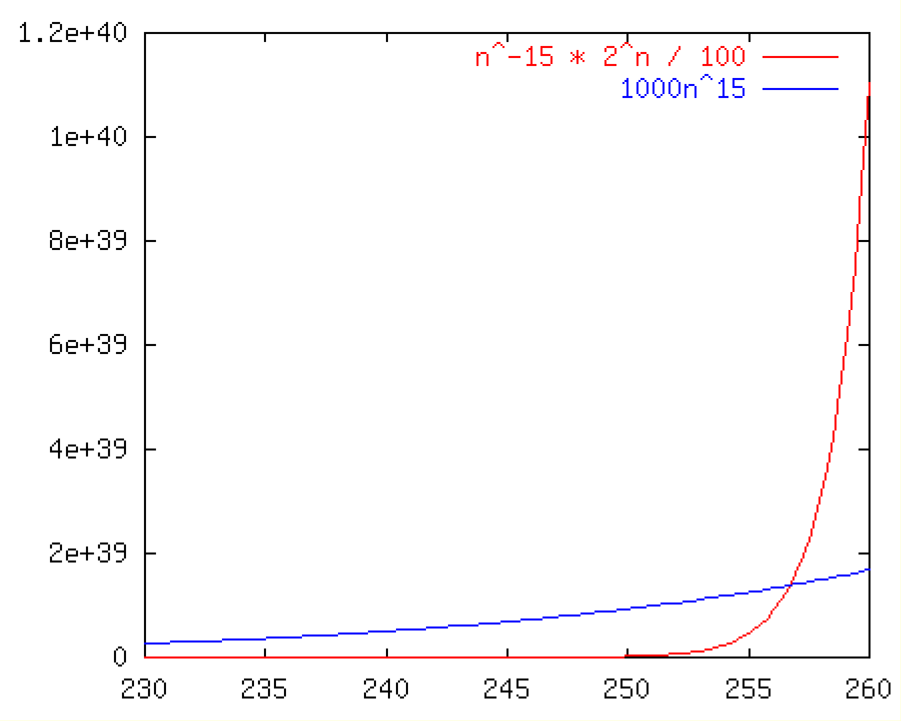




Race V

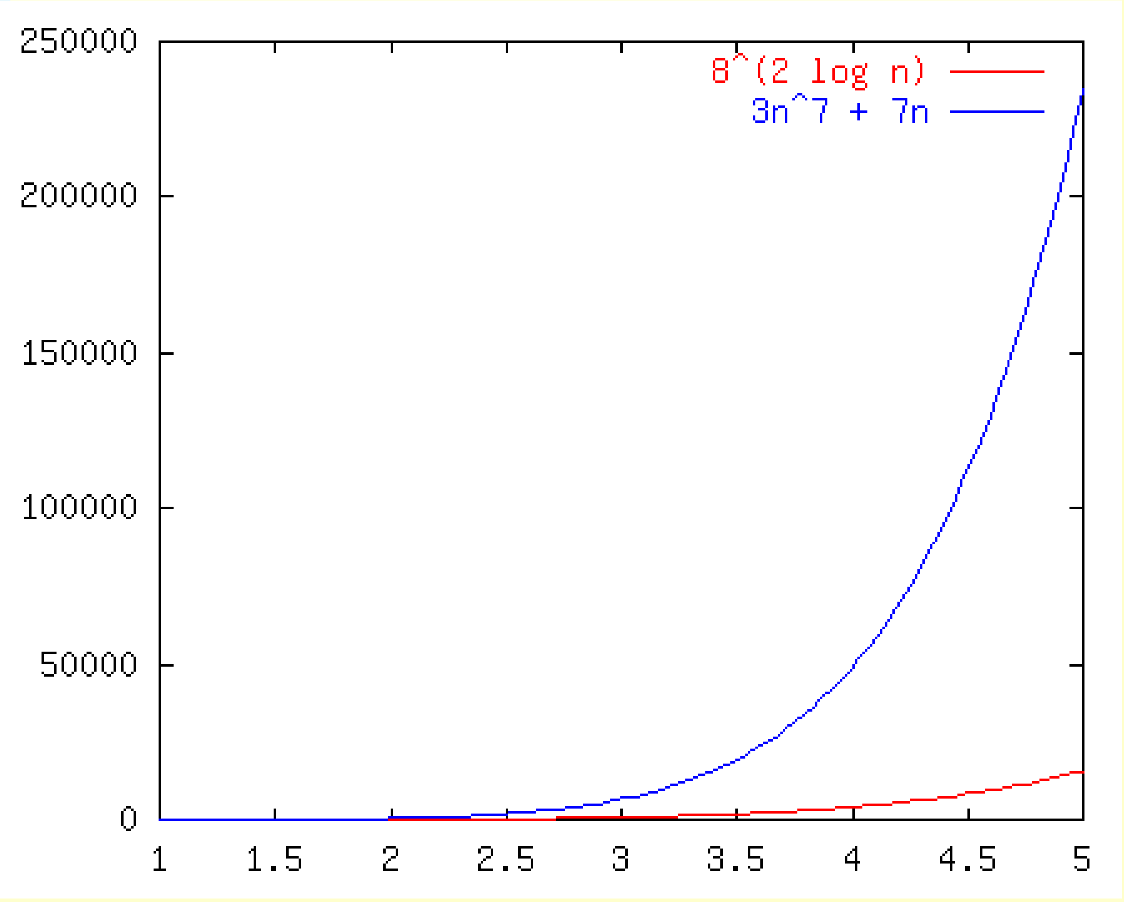
$n^{-15}2^n/100$ vs. $1000n^{15}$





Race VI

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



The Losers Win

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

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)

Note: More than one parameter, $O(\log m + 2^n)$ is not necessarily $O(2^n)$.

Performance Measurement

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

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

n	runTime (sec.)	n	runTime (sec.)
0	0	100	0
10	0	200	0
20	0	300	0
30	0	400	0
40	0	500	0
50	0	600	0
60	0	700	0
70	0	800	0
80	0	900	0
90	0	1000	0

Intel(R) Core(TM) i7-10710U CPU @ 1.10GHz 1.61 GHz, 16 GB RAM, Ubuntu 22.04.1 LTS

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.

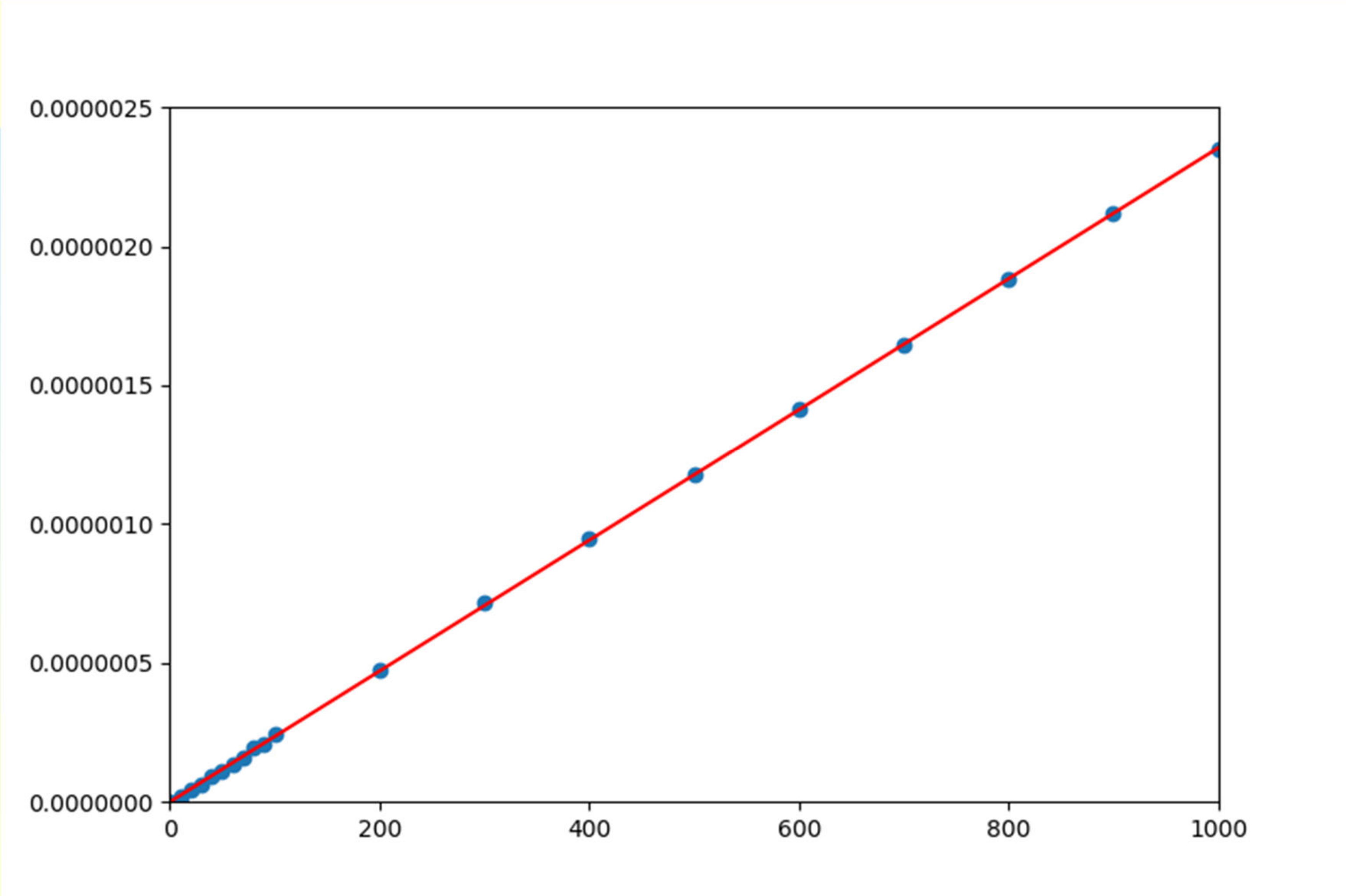
```
void TimeSearch ( )
{
  int a[1000], n[20];
  const long r[20] = {45000000000, 7500000000, 3000000000,
1800000000, 1650000000, 1200000000, 1000000000, 900000000,
700000000, 600000000, 450000000,300000000, 220000000,
130000000, 100000000, 80000000,70000000, 60000000,
50000000, 46000000};
  int  j;
  for (  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    totalTime    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 = SequentialSearch (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;
 }
}
```

n	totalTime (sec.)	runtime (sec.)	n	totalTime (sec.)	runTime (sec.)
0	145	3.22222e-09	100	111	2.46667e-07
10	151	2.01333e-08	200	143	4.76667e-07
20	124	4.13333e-08	300	158	7.18182e-07
30	116	6.44444e-08	400	123	9.46154e-07
40	147	8.90909e-08	500	118	1.18e-06
50	133	1.10833e-07	600	113	1.4125e-06
60	135	1.35e-07	700	115	1.64286e-06
70	144	1.6e-07	800	113	1.88333e-06
80	136	1.94286e-07	900	106	2.12e-06
90	124	2.06667e-07	1000	108	2.34783e-06

Intel(R) Core(TM) i7-10710U CPU @ 1.10GHz 1.61 GHz,
16 GB RAM, Ubuntu 22.04.1 LTS

runtime (sec.)



n

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

实验&作业

P69-73

- **实验：1，10**　**实验课上检查验收后提交源代码**
- **作业：3（必做题）；**

  4，5，6，7（选做2题）

  提交截止时间：10月7日晚22:00之前

注意：作业提交电子版，发送到助教邮箱，（可WORD/Latex编辑；也可手写拍照），建议文件格式为PDF，文件命名格式为“学号_姓名_第1章作业”