Example: *break* in Infinite Loops

- Write a program which calculates $log_{10}(x)$ with input x:
  - If $x > 0$, then output $log_{10}(x)$;
  - If $x = -1$, then exit the program;
  - Otherwise, display "Only positive numbers!" in the screen.
- Besides, the program terminates only when $x = -1$.

Input: x
Output: $log_{10}(x)$ if $x > 0$

Solution 1

```
clear all;
clc
% main
while 1
    x=input('Enter a number: ');
    if x>0
        y=log10(x)
    elseif x==-1
        break;
    else
        disp('Invalid input. Try again.');
    end
end
```

Solution 2

```
clear all;
clc
% main
x=input('Enter a number: ');
while x~=-1
    if x>0
        y=log10(x)
    else
        disp('Invalid input. Try again.');
    end
    x=input('Enter a number: ');
end
```

How many levels can one *break* break?[1]

- In nested loops, break exits from the innermost loop only.

```
clear all;
clc
% main
for i=1:5
    for j=1:i
        if i==3
            break;
        end
        fprintf('*');
    end
    fprintf('\n');
end
```

[1] Thanks to a lively class discussion (MATLAB-237) on April 19, 2014.

Exercise: *while* Loop By *for* Loop

- Redo the problem in p. 12 by replacing a while loop by a for loop and an if statement.

```
clear all;
clc
% main
s=0;
for i=1:inf
    s=s+5*i^2-2*i;
    if s>1e4
        break;
    end
end
i
s
```

Exercise

- Gauss' teacher asked him to add up all of the numbers 1 through 100 when he was 9.
- Question: $\sum_{i=1}^{100} i =?$

```
clear all;
clc
x = 0;
for i = 1:1:100
    x = x+i;
end
x
```

Figure : Carl Friedrich Gauss (1777–1855)

Extension

- Let n be an arbitrary positive integer given by the user.
- Let S_n be the sum of $1, 2, \ldots, n$.
- Show $S_1, S_2, \ldots, S_n$.
- Hint: You may need a nested *for* loop of two levels.

Solution 1

```
1  clear; clc;
2  % main
3  n = input('Please enter an positive integer: ');
4  y = zeros(1,n); % initialize an zero array of n ...
       elements
5  for i = 1:n
6      for j = 1:i
7          y(i) = y(i)+j;
8      end
9  end
```

Solution 2

```
1  clear; clc;
2  % main
3  n = input('Please enter an positive integer: ');
4  y = zeros(n,1); % init a zero array of n elements
5  x = 0; % init to zero
6  y(1) = 1;
7  for i = 2:1:n
8      y(i) = y(i-1) + i;
9  end
```

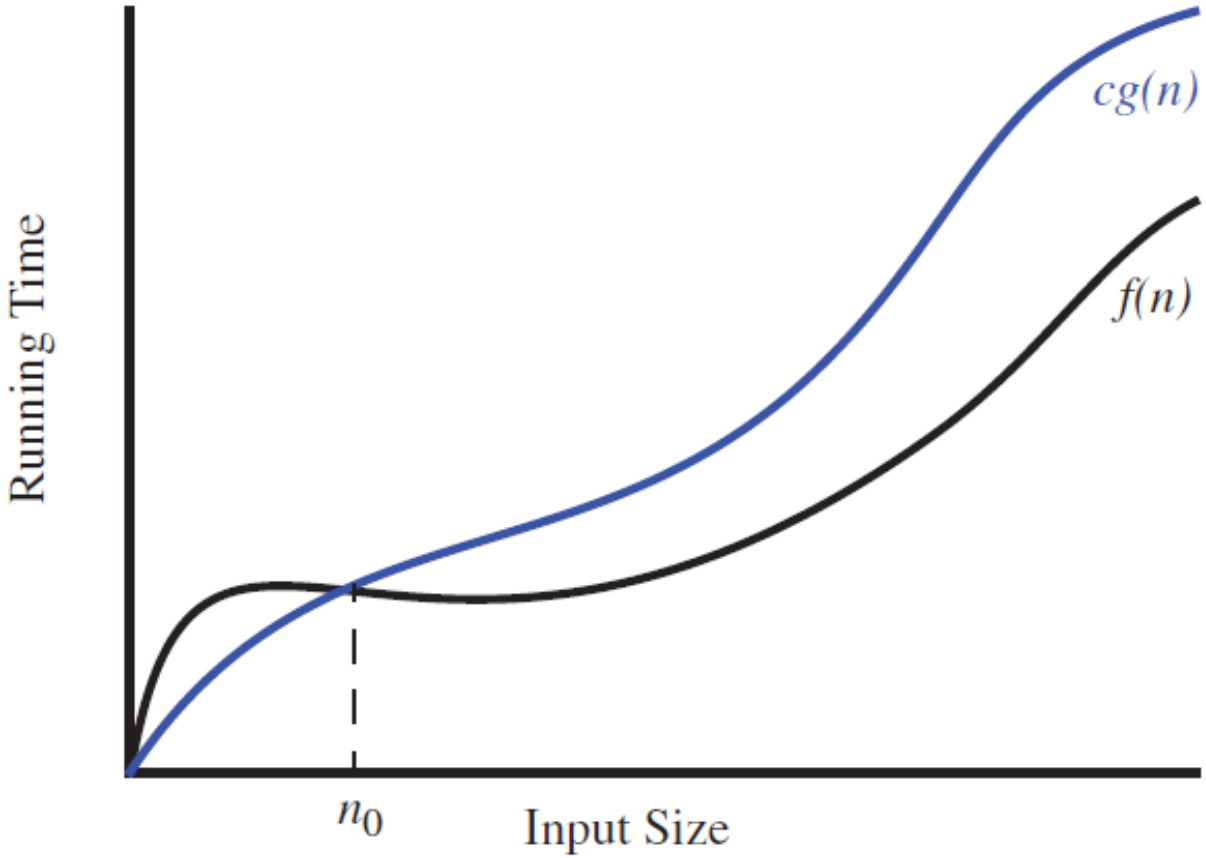
Analysis of Algorithms In Nutshell

- First, the algorithms for the same problem are supposed to be correct.
- Then we compare these algorithms.
- The first question is, Which one is more efficient? (Why?)
- In algorithm analysis, we focus on the growth rate of the running time or space requirement as a function of the input size n, denoted by $f(n)$.

O-notation

- In math, O-notation describes the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions.
- **Definition** (O-notation) $f(x) = O(g(x))$ as $x \to \infty$ if and only if there is a positive constant c and a real number x_0 such that

$$|f(x)| \leq c|g(x)| \quad \forall x \geq x_0. \tag{1}$$



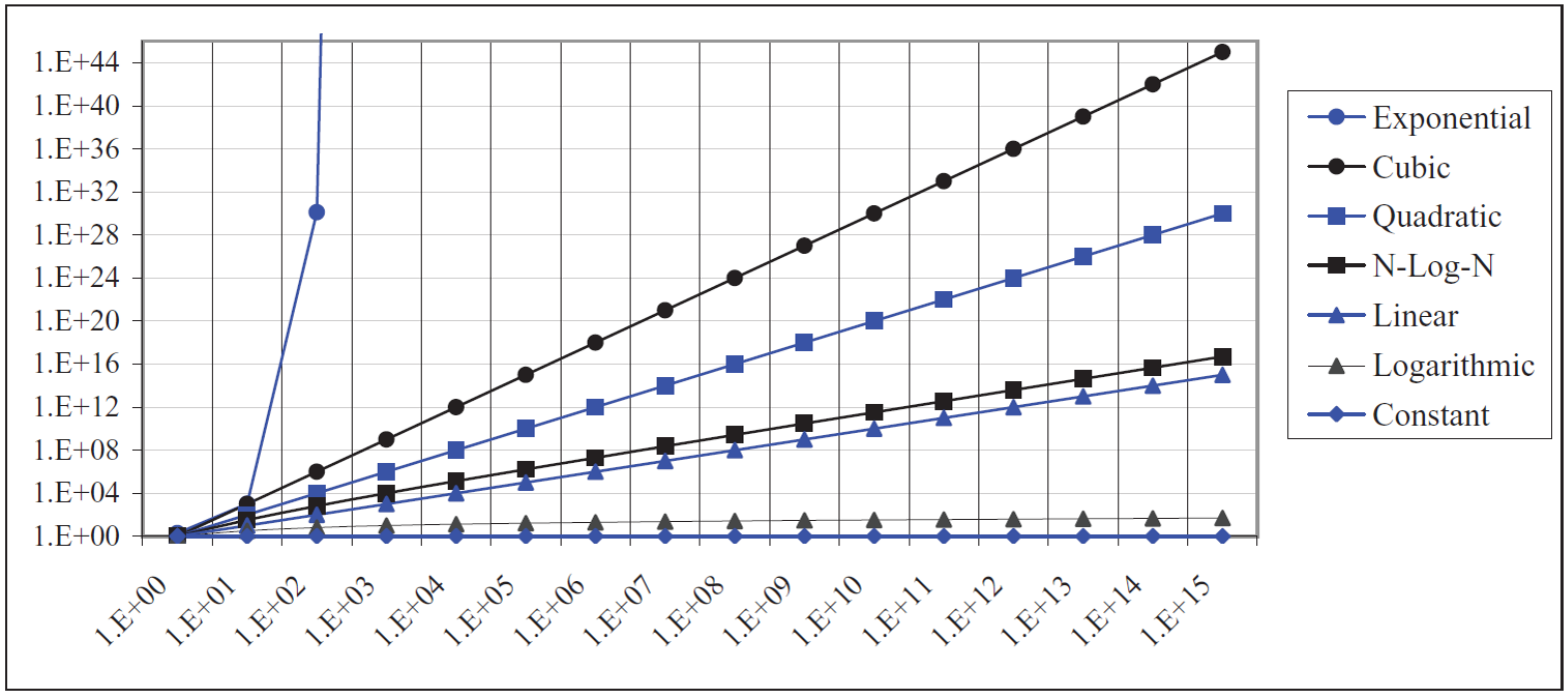
- For example, $8n^2 - 3n + 4 \in O(n^2)$.

- O-notation is used to classify algorithms by how they respond to changes in input size.[2]
  - Time Complexity
  - Space Complexity
- In short, O-notation describes the asymptotic[3] upper bound of the algorithm.
  - That is, the worst case we can expect.

[2] Actually, there are $\Theta, \theta, o, \Omega$, and ω which classify algorithms.

[3] The asymptotic sense is that the input size n grows toward infinity.

Growth Rates for Fundamental Functions[4]



constant	*logarithm*	*linear*	*n-log-n*	*quadratic*	*cubic*	*exponential*
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

[4]See Table 4.1 and Figure 4.2 in Goodrich and etc, p. 161.

Solution 1 vs Solution 2

- Time Complexity
  - *Solution 1* runs in $O(n^2)$ while *Solution 2* runs in $O(n)$.
  - So, we can say that *Solution 2* is more efficient[5] than *Solution 1*.
- Space Complexity
  - *Solution 1* and *Solution 2* both have $O(n)$.

[5] You can say that *Solution 2* runs faster than the other.

Closed-Form Formulae

- It's well-known that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- Moreover, it is also known that
  - $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$, and
  - $\sum_{i=1}^{n} i^3 = (\frac{n(n+1)}{2})^2$.
- How about $\sum_{i=1}^{n} i^k$ for any real number k?

Why Numerical Methods?

- Numerical methods open a window for many complicated problems, even if the correctness of the numerical solution remains a big problem itself.
- Simulation techniques significantly reduce the cost both in industry and science.
- Note that the problems with analytic solutions are rare.

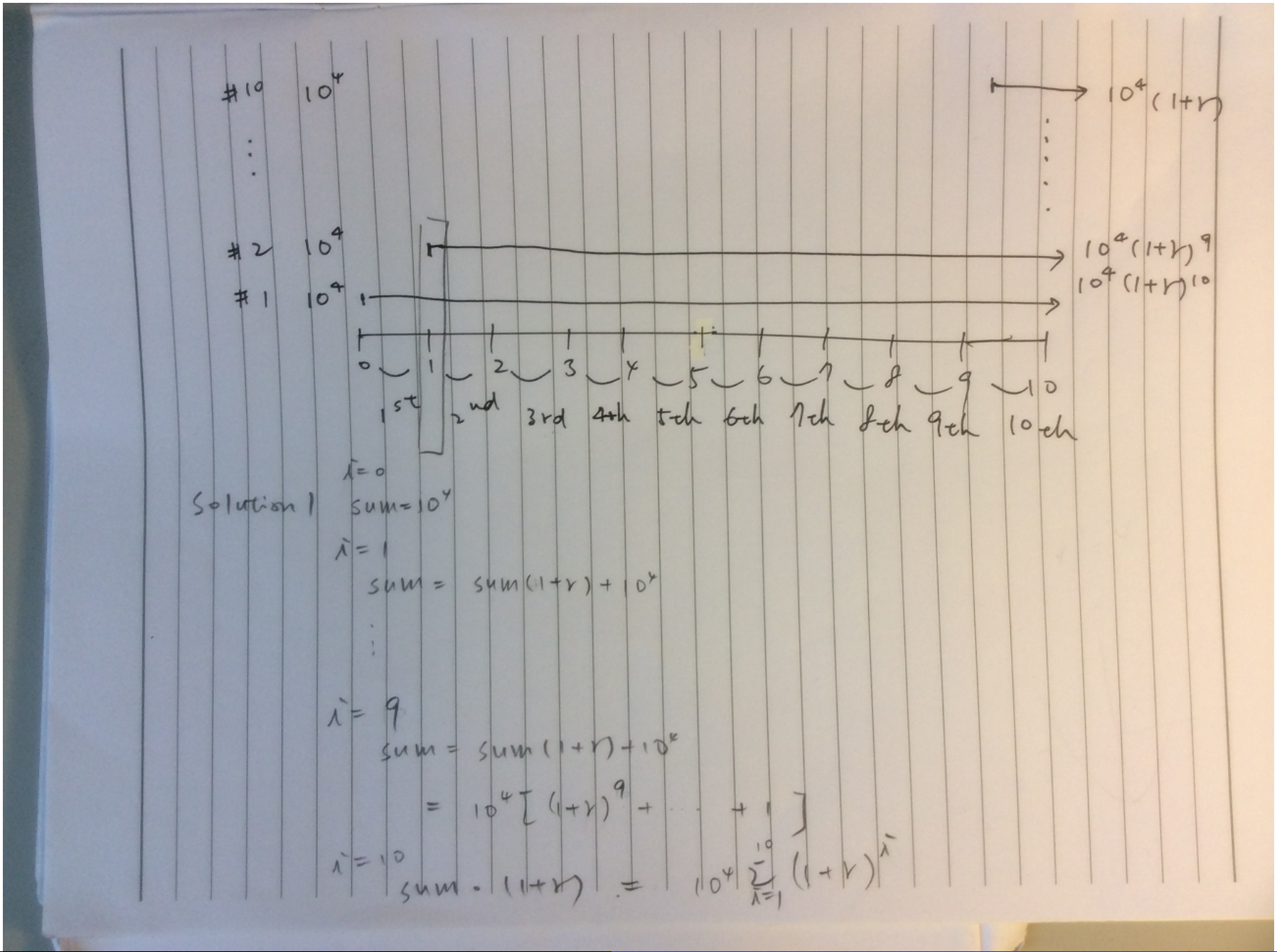
"All science is dominated by the idea of approximation."
– Bertrand Russell (1872-1970)

"Essentially, all models are wrong, but some are useful."
– George E. P. Box (1919-2013)

Exercise

- Consider a 10-year deposit account.
- You deposit 10,000 NTD in the beginning of each year[6].
- Assume that the annual compounding interest rate is $r = 10\%$.
- Determine the compound amount at the end of 10 years.
  - The answer is $175,311.670611$ NTD.

[6] So, $100,000$ NTD in total.


#10 10^4
#2 10^4
#1 10^4
$10^4(1+r)$
$10^4(1+r)^9$
$10^4(1+r)^{10}$
0 1 2 3 4 5 6 7 8 9 10
1st 2nd 3rd 4th 5th 6th 7th 8th 9th 10th
Solution 1
$i=0$ sum $=10^4$
$i=1$ sum $=$ sum$(1+r)+10^4$
$i=9$ sum $=$ sum$(1+r)+10^4$
$= 10^4[(1+r)^9 + \cdots + 1]$
$i=10$ sum$\cdot(1+r) = 10^4 \sum_{i=1}^{10}(1+r)^i$

Solution 1

```
clear all;
clc;
% main
format long g
s=1e4;
r=0.1; % Annual interest rate
for i=1:1:10
    if i~=10
        s=1e4+s*(1+r);
    else
        s=s*(1+r)
    end
end
```

- Time complexity: $O(n)$
- Space complexity: $O(1)$
- It is similar to Horner's method.

Solution 2

```
clear all;
clc;
% main
format long g
s=0;
r=0.1;
i=1;
while (i<=10)
    s=s+(1+r)^i;
    i=i+1;
end
s=1e4*s
```

- Time complexity: $O(n)$
- Space complexity: $O(1)$

Solution 3

```
clear all;
clc;
% main
format long g
x=1e4;
r=0.1;
n=10;
s=x*(1+r)*((1+r)^n-1)/r
```

- Time complexity: $O(1)$ (Why?)
- Space complexity: $O(1)$
- Remarks:
  - Pros: Fastest and compact code.
  - Cons: Less flexible.

Extension: Floating Interest Rate

- Instead of constant interest rate, the floating interest rates are given by

$$r = 0.1 : -0.01 : 0.01.$$

  Then determine the compound amount of the previous example.

- $125,743.635988833$ NTD

Solution

```
clear all;
clc;
% main
format long g
s=1e4;
r=0.1:-0.01:0.01; % Annual interest rate
for i=1:1:10
    if i==10
        s=s*(1+r(i))
    else
        s=1e4+s*(1+r(i));
    end
end
```

Always Positive Interest Rate?

- Draghi Unveils Historic Measures Against Deflation Threat (Jun 6, 2014 12:07 AM GMT+0800)
  - The ECB today cut its deposit rate to minus 0.1 percent, becoming the first major central bank to take one of its main rates negative.
- ECB降息市場反應激烈 歐元貶破1.3美元 (2014.09.05 04:13 am)
  - “...銀行存放央行的存款利率從負0.1%降至負0.2%...”

For Your Reference

- 10 Programming Tips For Beginners
  - If your code doesn't work, don't take it personally.
  - Google is your helper.
  - Reading error messages.
  - Why and what you declaring?
  - Construct your problem as per behavior.
  - Take a print of the statements!
  - Small issues can create bigger issues.
  - Switch off the monitor.
    - Writing code must be the final step in finding solution to a problem and not the first.
  - The computer will do what you instruct it to do, in the order you ask it to do.
  - If you have a TA or a friend offering help, please take it!

Vectorization[7]

- MATLAB is optimized for array operations.
- The built-in MATLAB functions such as sqrt(x) and exp(x) automatically operate on array arguments to produce an array result with the same size as the array argument x.

```
>> t=linspace(0,pi,5)
>> y=sin(t)

y =

         0    0.7071    1.0000    0.7071    0.0000
```

[7] More about vectorization

Advantages

- Appearance: Vectorized mathematical code appears more like the mathematical expressions found in textbooks, making the code easier to understand.
- Less error prone: Without loops, vectorized code is often shorter.
  - Fewer lines of code mean fewer opportunities to introduce programming errors.
- Performance: Vectorized code often runs much faster than the corresponding code containing loops.

Performance Analysis In Real Time

- In addition to the theoretical analysis of algorithms[8], programmers can also use a timer to measure performance.
- Note that the results may differ depending on the difference of run-time environments.
- Make sure that you benchmark the algorithms on the same conditions.
- Once you identify which functions are consuming the most time, you can determine why you are calling them.
- Then, look for ways to minimize their use and thus improve performance.

[8] Recall the O-notation.

Example: **tic-toc**

- **tic** starts a stopwatch timer.
- **toc** reads the elapsed time from the stopwatch timer started by **tic**. (Try.)
  - Pros: More precise, $\approx 10^{-6}$ second.
  - Cons: Returns values only in second;

```
1 >> tic % Please wait for a second.
2 >> toc
3
4 Elapsed time is 2.279756 seconds.
```

Tips To Improve Performance

- (Preallocate arrays) Repeatedly resizing arrays often requires MATLAB to spend extra time looking for larger contiguous blocks of memory, and then moving the array into those blocks.
- (Vectorize your code) Take advantage of the full functionality of MATLAB.
  - Element-by-element operations do help increase the performance.
- (Exploit functions) Construct separate functions from the larger chunks of code.
  - Use functions instead of scripts because they are generally faster.
- (Avoid overloading built-in functions) Overloading may negatively affect performance.

Example: Dynamic Allocation Is Poor

```
1  clear all;
2  clc
3  % main
4  tic
5  i=1;
6  for t=0:.01:100
7      y(i)=sin(t); % dynamic allocation of array y
8      i=i+1;
9  end
10 toc
11
12 clear y;
13 tic
14 t=0:.01:100;
15 y=sin(t); % vectorization
16 toc
```

```
1 Elapsed time is 0.069676 seconds.
2 Elapsed time is 0.000312 seconds.
```

- This speedup[9] is around 223.3.
- Note that this number may be different computer by computer.

[9] The speedup is given by t_1/t_2 where t_1 and t_2 are time costs.

Element-By-Element Operations

Symbol	Operation	Form	Example
+	Scalar-array addition	A + b	[6,3]+2=[8,5]
−	Scalar-array subtraction	A − b	[8,3]−5=[3,−2]
+	Array addition	A + B	[6,5]+[4,8]=[10,13]
−	Array subtraction	A − B	[6,5]−[4,8]=[2,−3]
.*	Array multiplication	A.*B	[3,5].*[4,8]=[12,40]
./	Array right division	A./B	[2,5]./[4,8]=[2/4,5/8]
.\	Array left division	A.\B	[2,5].\[4,8]=[2\4,5\8]
.^	Array exponentiation	A.^B	[3,5].^2=[3^2,5^2]
			2.^[3,5]=[2^3,2^5]
			[3,5].^[2,4]=[3^2,5^4]

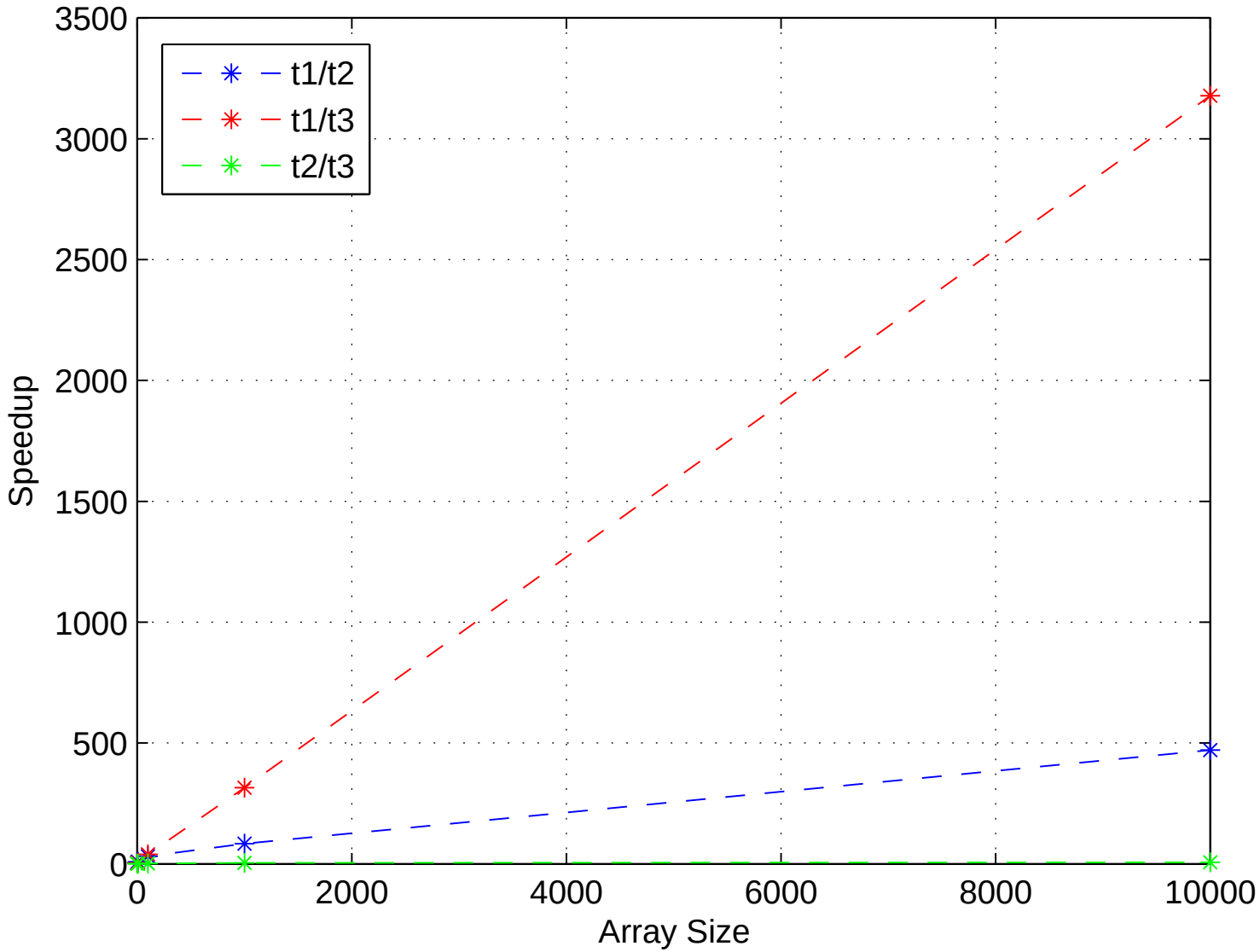
- Note that $A. * B$ is not equivalent to the inner product of A and B.
- We will see the difference between left and right divisions in matrix computation.

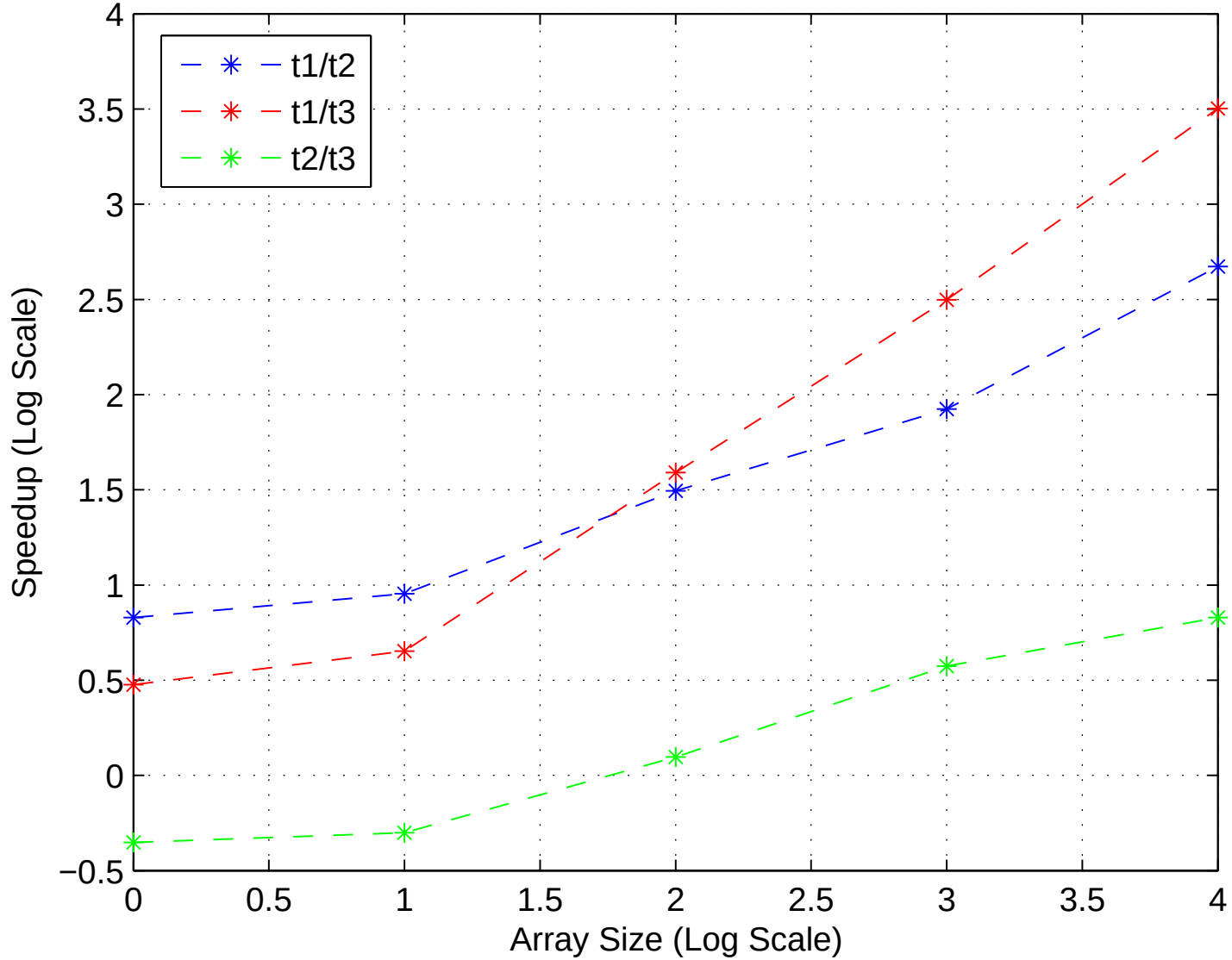
Example: Vectorization

```
1  clear all;
2  clc
3  % main
4
5  order=0:1:4;
6  t1=zeros(length(order),1);
7  t2=zeros(length(order),1);
8  t3=zeros(length(order),1);
9
10 for j=1:1:length(order)
11
12     tic
13     for i=0:1:10^order(j)
14         y(i+1)=i^2; % dynamic allocation of array y
15     end
16     t1(j)=toc;
17
18     clear y;
```

```
19      array_len=length(0:1:10^order(j));
20      y=zeros(array_len,1); % preallocation of ...
            array y
21      i=1;
22      tic
23      for i=0:1:10^order(j)
24          y(i+1)=i^2;
25      end
26      t2(j)=toc;
27
28      clear y;
29      t=0:1:10^order(j);
30      tic
31      y=t.^2; % vectorization
32      t3(j)=toc;
33  end
34  figure(1),plot(10.^order,(t1./t2),'b*:',...
35          10.^order,(t1./t3),'r*:',10.^order,(t2./t3),'g*:');
36  legend(['t1/t2';'t1/t3';'t2/t3']);
```

```
37 ylabel('Speedup');
38 xlabel('Array Size');
39 figure(2),plot(order,log10(t1./t2),'b*:',...
40       order,log10(t1./t3),'r*:',order,log10(t2./t3),'g*:');
41 legend(['t1/t2';'t1/t3';'t2/t3']);
42 ylabel('Speedup (Log Scale)');
43 xlabel('Array Size (Log Scale)');
```


t1/t2
t1/t3
t2/t3
Speedup
Array Size
0
500
1000
1500
2000
2500
3000
3500
0
2000
4000
6000
8000
10000


t1/t2
t1/t3
t2/t3
Speedup (Log Scale)
Array Size (Log Scale)
4
3.5
3
2.5
2
1.5
1
0.5
0
−0.5
0
0.5
1
1.5
2
2.5
3
3.5
4