COMP 2310
Data Structures and Algorithms

Lecture 2

Algorithm Complexity

Mathematical Reminders

Exponents, Summations, Logarithms

Complexity Classes

Common Functions: Exponents

Polynomials:

$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$

Exponentials:

$a^0 = 1$

$a^1 = a$

$a^{-1} = 1/a,$

$(a^m)^n = a^{mn}$

$a^m a^n = a^{m+n}$

Summations

$$\sum_{i=1}^{n} ca_i = c\sum_{i=1}^{n} a_i$$

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

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

$$\sum_{i=1}^{n} i^k = \Theta(n^{k+1}) \ldots$$

Floor and Ceiling Functions

Let x be a real number.

- The floor of x, denoted by $\lfloor x \rfloor$, is defined as the greatest integer less than or equal to x.
- The ceiling of x, denoted by $\lceil x \rceil$, is defined as the least integer greater than or equal to x.
- For *x* an integer, $\lfloor x \rfloor = \lceil x \rceil = x$

  Examples:

  $\lfloor 5.4 \rfloor = 5$, $\lfloor -6.4 \rfloor = -7$,

  $\lceil 5.4 \rceil = 6$, and $\lceil -6.4 \rceil = -6$

Common Functions: Logarithms

The logarithm of a number to a given base is the power to which the base must be raised in order to produce that number.

We know that 10^3 =1000, what is 3 here?

$$\log_{\underset{base}{\underset{\uparrow}{10}}} 1000 = \underset{\log of\, 1000}{\underset{\uparrow}{3}}$$

Similarly, since 2^6 = 64, $\log_2 64$ = 6

Common Functions: Logarithms

Three particular values for the base *b* are most common:

The natural logarithm **(**ln**):** Base *b* = e = 2.71828…
The common logarithm **(**log**):** Base *b* =10
The binary logarithm **(**lg or log**) :** Base *b* = 2

In computer applications we use log base 2.

So, log 32 means $\log_2 32$

Common Functions: Logarithms

To find the logarithm of a number with a given base we solve the following equation

$$b^x = a$$

Examples:

$$10^x = 1000 \;\Rightarrow x = 3$$

$$2^x = 16 \;\Rightarrow x = 4$$

$$e^x = 2.71828 \;\Rightarrow x = 1$$

Common Functions: Logarithms

Some useful expressions:

$$\log(xy) = \log x + \log y$$
$$\log(x/y) = \log x - \log y$$
$$b^{\log_b a} = a$$
$$\log_b a^n = n \log_b a$$

$$\log_b(1) = 0$$
$$\log_b(b) = 1$$

Algorithm Complexity

Algorithm Complexity

- Algorithms differ in efficiency.

- The difficulty of algorithms for comparing their efficiencies is called computational complexity.

- Computational complexity indicates how much effort is required for an algorithm or how costly it is.

- An algorithm is hardly of any use if
  - very long time
  - large main memory

How to Measure Complexity ?

The cost of an algorithm can be measured via:

- Time (CPU usage)
- Space (Memory and disk usage)
- Network usage

The time factor is generally more important than the others.

Time Factor - How to measure time cost of an algorithm?

Can we use physical run time as a criterion?

No. Because run time is always system dependent

To make a meaningful comparison, all algorithms must run on the same machine.

Run time of an algorithm also depends on the operating system and language used to implement it.

Physical run time is not a good measure of complexity.

Finding a Complexity Measure

- We do not use real-time units such as micro seconds.
- We use logical units instead of real-time units.
- We need a functional relationship between the size of an object (array, file…) and a logical time measure for the algorithm.

Examples: Assume n is the size. Relationships between execution time t and n may be:

$$t = cn$$
$$t = a \log_2 n$$
$$t = an^3 + bn^2 + \ldots.$$

…………………………

where a, b and c are some constants.

Finding a Complexity Measure

How to determine a good functional relationship?

- A function expressing the relationship between ***t*** and ***n*** usually is not in a simple form, but :
  - Finding such a function is important only for large data sizes.
  - We can usually ignore some of the terms that do not substantially change the function's magnitude.
- The resulting function is simple and an approximation of the original function.
- For very large n, this approximation is satisfactory.
- The simplified measure of efficiency is called asymptotic complexity of the algorithm.

Growth of Functions: Most Influential Term

Example: Growth of a function

The growth rate of the terms in the function

$$f(n) = n^2 + 10n + 100$$

n	n^2	$10n$	100	$f(n)$	Contribution to $f(n)$ by n^2
1					
10					
100					
1000					

Growth of Functions: Most Influential Term

Example: Growth of a function

The growth rate of the terms in the function

$$f(n) = n^2 + 10n + 100$$

n	n^2	$10n$	100	$f(n)$	Contribution $f(n)$ by n^2
1	1	10	100	111	% 0.9
10	100	100	100	300	%33.3
100	10,000	1000	100	11,100	%90.1
1000	1,000,000	10,000	100	1,010,100	%99.0

Most influential term in f(n) is n^2

Simplified Expression: Big-O notation

We need a simplified expression to represent the growth of a function.

Definition: Big-O

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

if there exists a positive integer n_0 and a positive constant c, such that

$f(n) \leq c.g(n) \quad \forall\, n \geq n_0$

Note that, since there are 2 unknowns, the solution will not be unique.

Many different pairs of n_0 and c will satisfy the inequality.

Simplified Expression: Big-O notation

Example:

Show that $f(n)=3n+8$ is $O(n)$

We need to show that $3n+8 <= cn$

$\rightarrow (c-3)n >= 8$, Take $c=4$

$\rightarrow$ $n >= 8$, so we have a solution for $c=4$ and $n_0=8$

Big O : Example-1
How to Define c?

$f(n) = 2n + 5$
$g(n) = n$

- Consider the condition

  $$2n + 5 <= n$$

  will this condition ever hold? No!

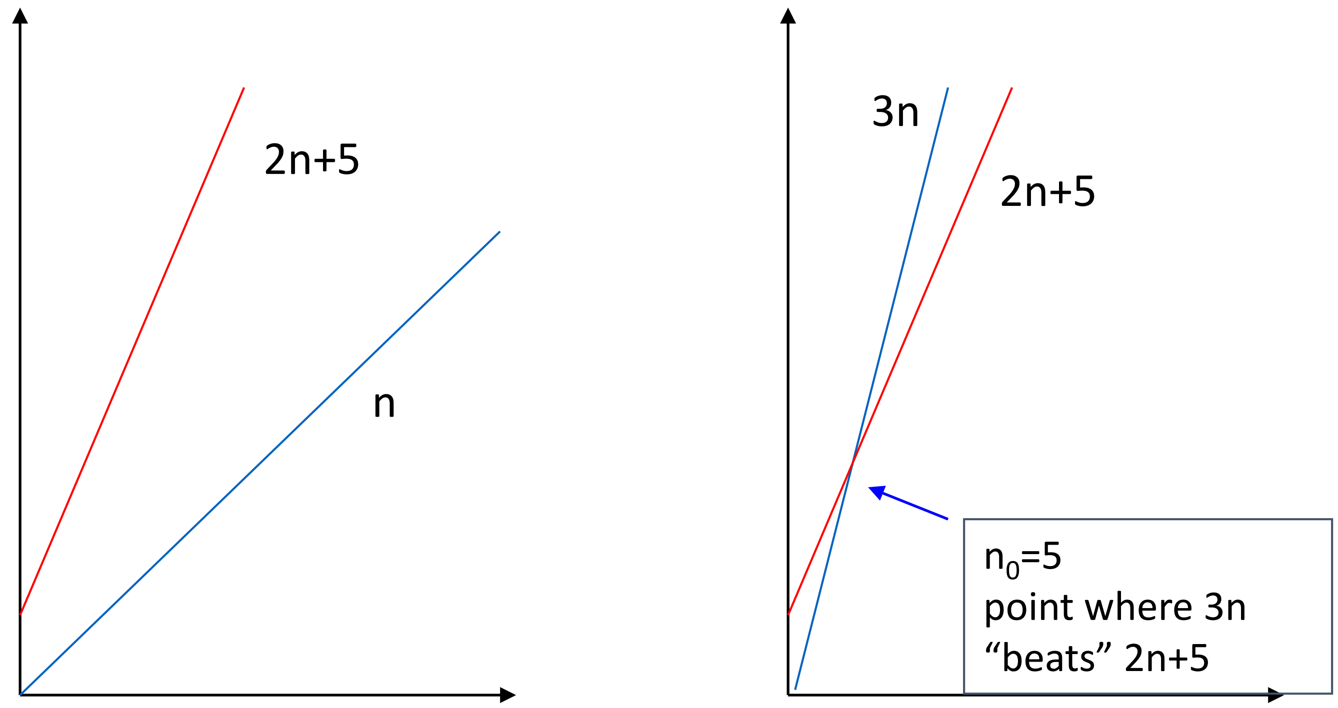
- How about if we stick a constant to n?

  $$2n + 5 <= 3n$$

  the condition holds for values of n greater than or equal to 5

- This means we can select $c = 3$ and $n_0 = 5$

Example-1 : Illustration



2n+5 is O(n)

Big-O notation : Example2

Show that, $2n^2+4n+1$ is $O(n^2)$

We need to show that the condition

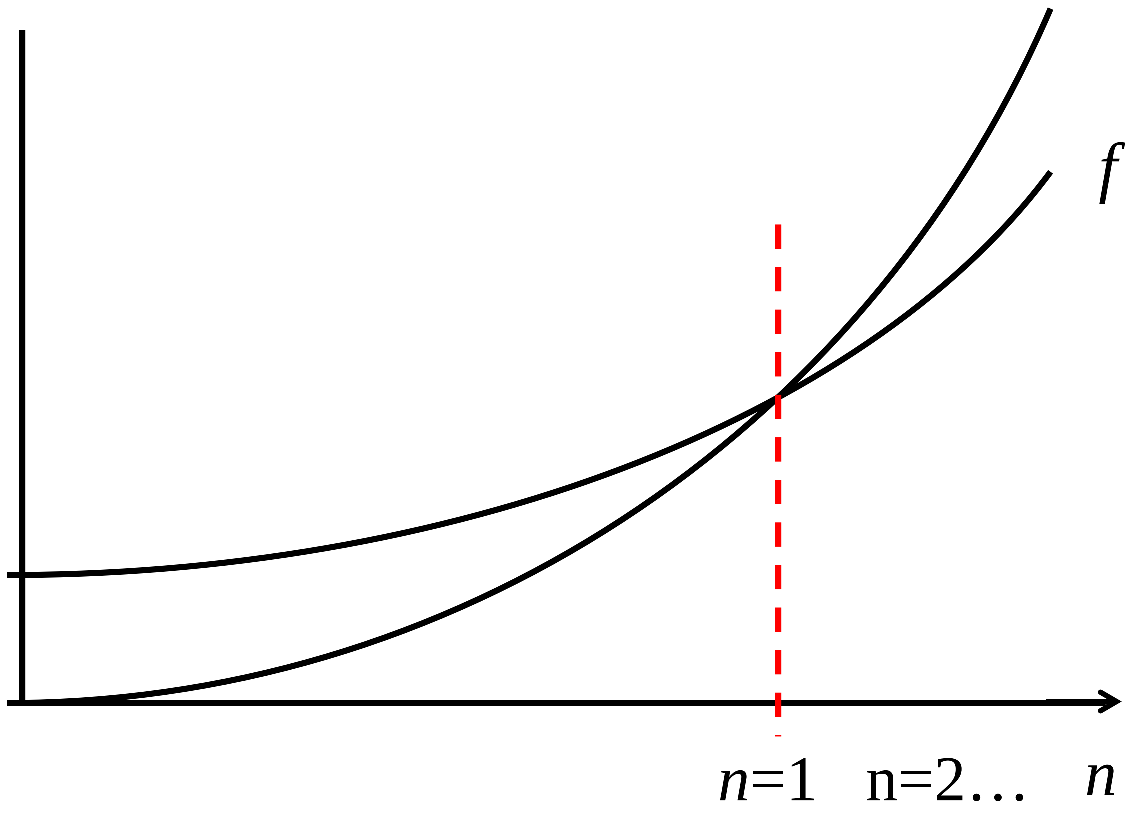
$$2n^2 + 4n + 1 \leq cn^2$$

is satisfied for some c and n. For example, take $n=1$. We have

$$2n^2 + 4n + 1 \leq cn^2$$
$$2 + 4/n + 1/n^2 \leq c$$
$$2 + 4/1 + 1/1 \leq c$$
$$c \geq 7$$

This is not a unique solution. The values for n and c satisfy the inequality in the definition. But many many different pairs will also satisfy the inequality.

Big-O notation: Illustration-2



Simplifying Big-O Notation

We usually need to use the simplest formula in the Big-O notation.

- We write

$$5n^2 + 2n + 3 = O(n^2)$$

- The following expressions are all correct but we use the final form:

$$5n^2 + 2n + 3 = O(5n^2 + 2n)$$
$$5n^2 + 2n + 3 = O(5n^2)$$
$$5n^2 + 2n + 3 = O(n^2)$$

Simplifying Big-O Notation

Examples:

$$f(n) = 5n^2 + 2n = O(n^2)$$

$$f(n) = 3n \log n - 2 = O(n \lg(n))$$

$$f(n) = \log n + 2\log(\log n)) = O(\log n)$$

$$f(n) = n + \log n^n = O(n \log n)$$

Asymptotic Bounds for Polinomials

Polynomials: For $m > 0$
$f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0 = O(n^m)$

The highest exponent determines the complexity class.

Example :

$f(n) = 15n^2 + 199n + 12000$
$= O(n^2)$

Determining Asymptotic Complexity

- f(n) should represent runtime of an algorithm closely. How to determine it?
- We have to count the number of basic operations such as assignments, arithmetic operations when their numbers grow.

Example 1: Consider the code fragment to calculate

$$T = \sum_{i=1}^{n} i^3$$

```
int sum3(int n)
{  int sum;
   sum=0;
  for(int i=1,i<=n, i++)
        sum +=i*i*i;
   return sum;
}
```

Determining Asymptotic Complexity

How many basic operations should be performed?

Ignore declarations.

```
                                              no. of opns
int sum3(int n) ..........                        0
{                              ..........         0
      int sum;                 ..........         0
      sum=0;                   ..........         1
      for(int i=1,i<=n ,i++)        ...           2n+2
         sum +=i*i*i;            ........         4n
      return sum;               ..........        1
}                                  .........      0
Total # of operations:              .....         6n+4
```

(1 for initialize, n+1 for the tests, n for the increments)

(four operations that are executed: +=**)

Determining Asymptotic Complexity

Example Explained:

We assume that declarations need no time.

Line 1: Counts 1.
Line 2: Initializing *i* counts 1,
testing for *i*<=*n* counts *n*+1,
incrementing *i* counts *n*.
Line 3: Two multiplication counts 2*n,*
one addition couns *n,*
one assignment counts *n,*
Line 4: Counts 1.
Total : f(n)= 6*n*+4 which is *O*(*n*).

General Rules

Rule 1:

Running time of "for" loops = Running times of the statements inside the loop x the number of iterations

Example :

```
for (i=0; i<n; i++)
     k++;
```

The code is $O(n)$

General Rules

Rule 2:

Running time of nested loops =The running time of the statements inside the loop x The product of the sizes of all the loops.

Example :

```
for (i=0; i<n,i++)
  for(j=1,j<=n,j++)
        k++;
```

The code is $O(n^2)$

Note that we ignore non influential terms and even coefficients in actual f(n)

Finding Asymptotic Complexity

More than one loops :

```
for (i=0; i<n; i++)        O(n)
   a[i]=0;
for (i=0; i<n; i++)
  for (j=0; j<n; j++)      O(n^2)
     a[i]+=a[j]+i+j;
```

Asymptotic complexity:

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

General Rules

Rule 3:

Running time of control statements = The running time of the statements of the test + maximum of the running times of the conditions

Rule 4:

Time complexity of a function call (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

Example :

```
a = ARR[5];
```

Complexity is independent of the input size.

General Rules

Rule 5: O(logn)

Time Complexity of a loop is considered as O(logn) if the loop variables is divided(/) or multiplied(*) by a constant amount.

Example: How many times (k) the loop will be executed?

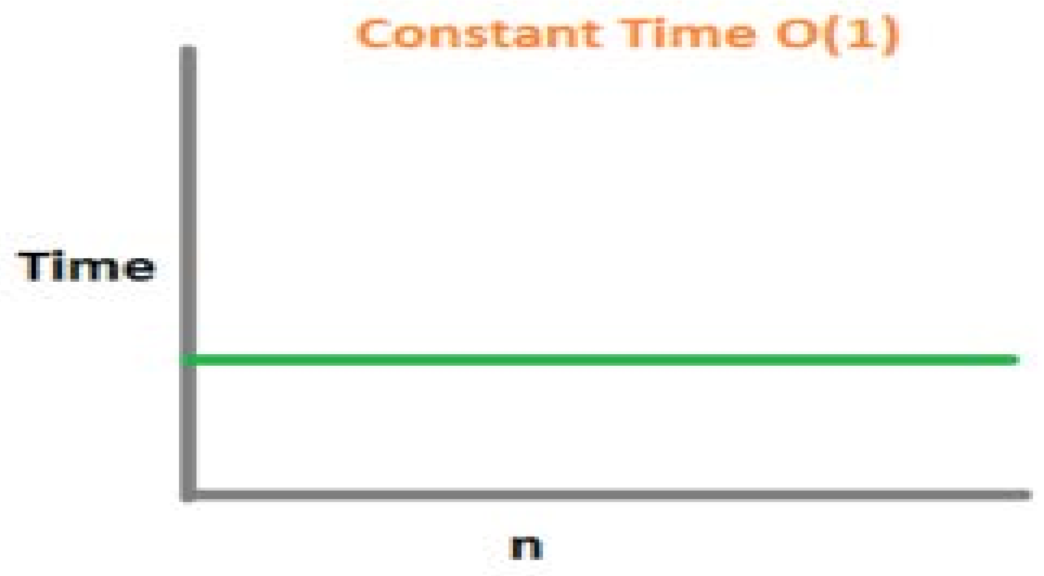
n=32;
k=0;
for(i=n, i>1, i=i/2)   // Example : i=32,16,8,4,2,1.
    k++; // $k = 5 = \log_2 32$

....

k=0;
for(i=1, i<n, i=i*2) // i=1,2,4,8,16,32, k=5
    k++;

Constant Time

Constant time implies that the number of operations the algorithm needs to perform to complete a given task is independent of the input size.



Examples :

- Accessing a single element of an array.
- Given a list, report the first element

Worst Case – Best Case - Average Case

Worst case Running Time: The behavior of the algorithm with respect to the worst possible case of the input instance

An upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer

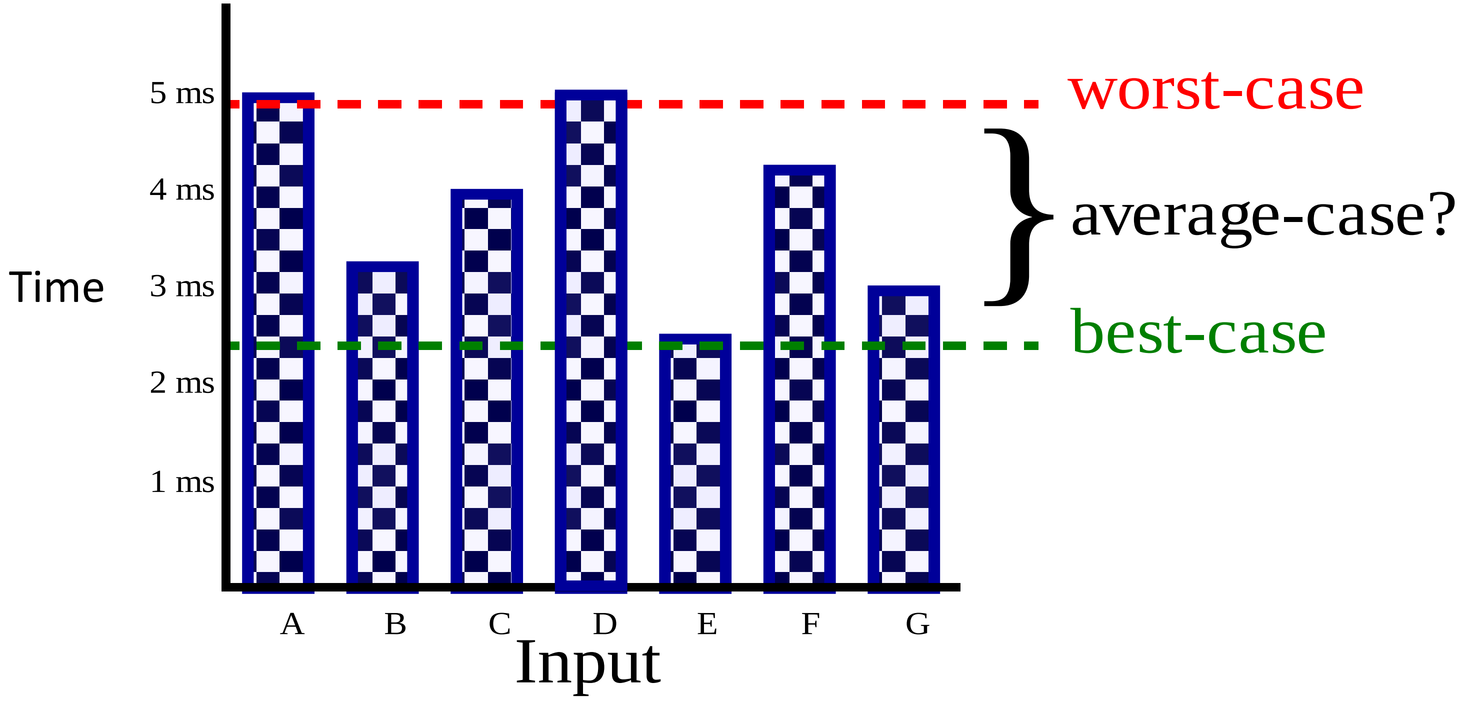
Best case Running Time: The input of size n for which the algorithm runs the shortest among all possible inputs of same size. Lower bound on running time of an algorithm. Not very informative!

Average case Running Time: An estimate of the running time for an "average" input. It is assumed that all inputs of a given size are equally likely.

The average case analysis is not easy to do in most practical cases and it is rarely done.

→In general, we perform worst case analysis .

Worst Case – Best Case - Average Case



Typically algorithms complexities are measured by their *worst case*

Worst Case – Best Case - Average Case

Consider the following algorithm for finding the position of a given value in an array:

What are the best, worst and average case complexities?

FIND(A, k)
$\rhd$ Finds a given value in a given array
$\rhd$ Output: The index of the firstly found element of A
that matches k or -1.
$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq k$ **do**
    $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

Worst Case – Best Case - Average Case

We have three questions about runtime of Find():

1. How long will it take in the best case?

   In the best case, the target value is the first element of the array.

   So the search takes $O(1)$

2. How long will it take in the worst case?

   In the worst case, the value is the last element of the array or it is not in the array.

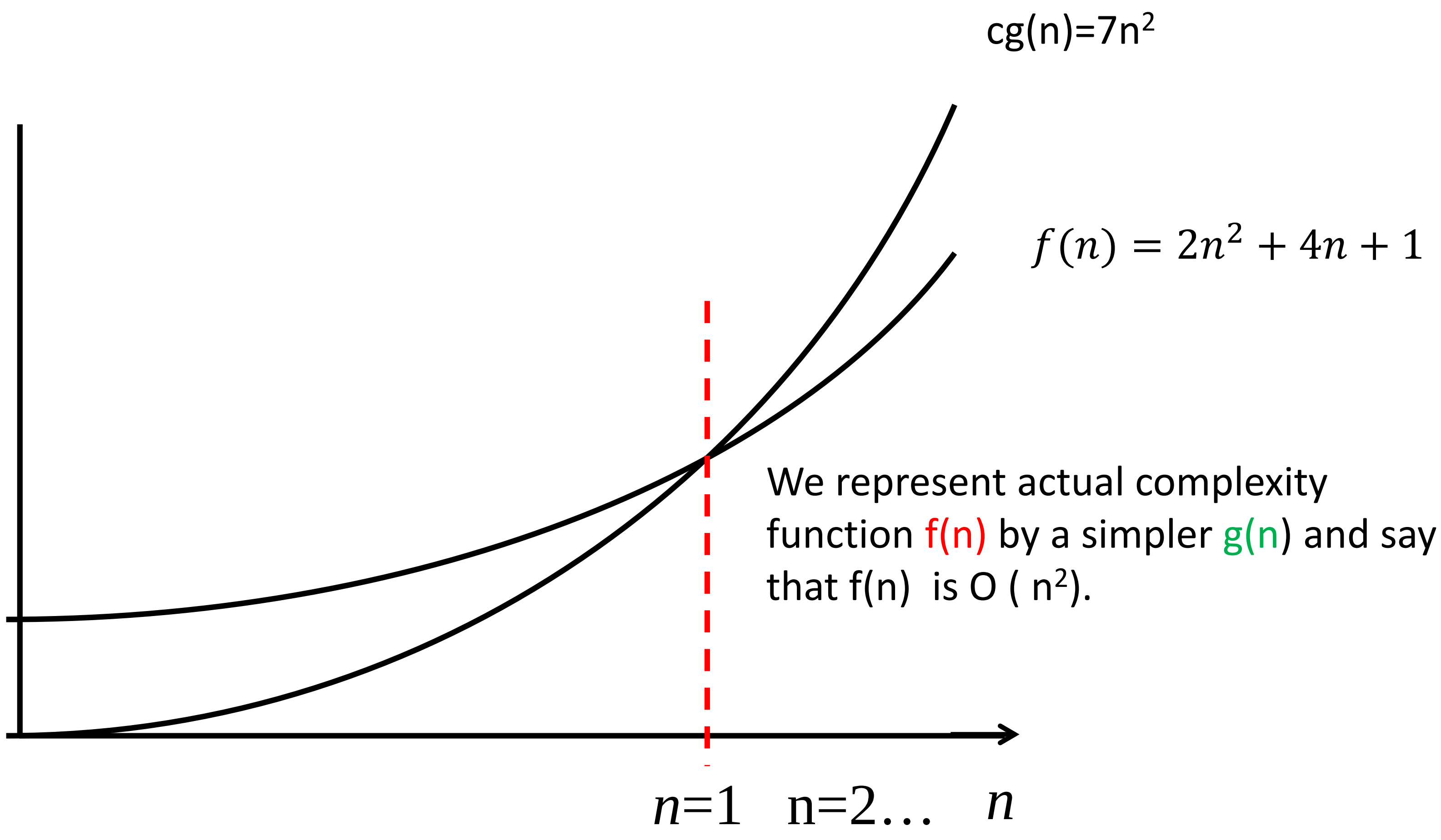
   So the search takes time proportional to the length of the array: $O(n)$.

3. How long will it take in the average case?

   Any position in the array is equally likely.

   So the expected value of the position will be $(n/2) = O(n)$

Big-O notation: Reminder



We represent actual complexity function f(n) by a simpler g(n) and say that f(n) is O (n^2).

Complexity Order of Common Functions

Constant<Logarithmic<linear<linear*log<quadratic<cubic<....<exponential

$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < \ldots\ldots < O(c^n)$, $(c>1)$

Polinomial time algorithms:

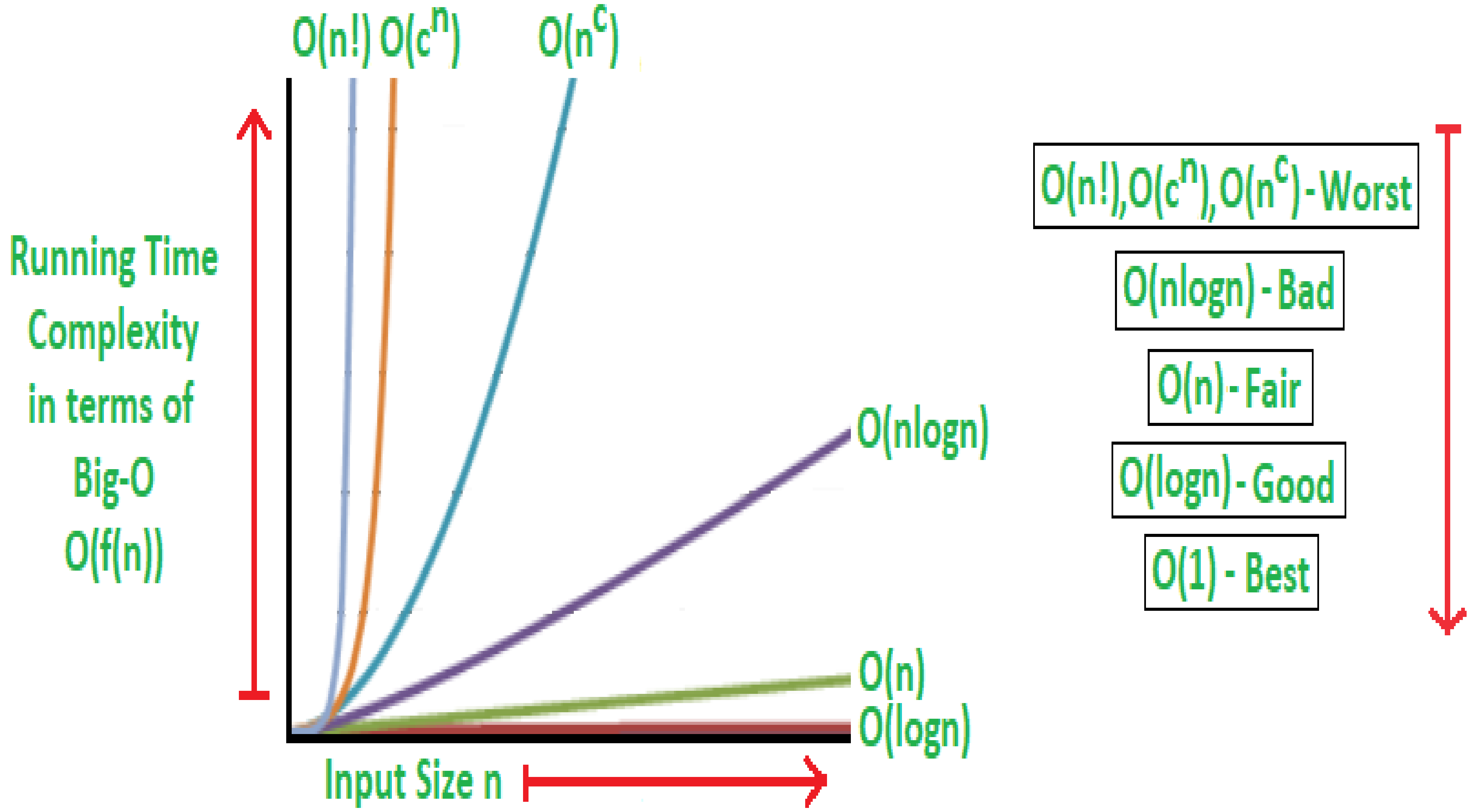
$O(n^k)$, $k>= 1$

Exponential time algorithms :

$O(c^n)$, n represents power.

Growth of Complexity Functions

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(n^3)	O(2^n)
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}



Complexity functions : n is the input size and c is a positive constant.

Running Times

- Assume N = 100,000 and processor speed is 1,000,000,000 operations/sec

Times needed for various complexity functions:

Function	Running Time
2^N	$3.2 \times 10^{30,086}$ years
N^4	3171 years
N^3	11.6 days
N^2	10 seconds
$N\sqrt{N}$	0.032 seconds
N log N	0.0017 seconds
N	0.0001 seconds
$\sqrt{N}$	3.2×10^{-7} seconds
log N	1.2×10^{-8} seconds

Common Time Complexities

BETTER

- $O(1)$ constant time
- $O(\log n)$ log time
- $O(n)$ linear time
- $O(n \log n)$ log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(n^k)$ polynomial time
- $O(2^n)$ exponential time
- $O(n!)$ “
- …….

WORSE

Exponential Time?

What is the implication of exponential time?

- Estimated age of the Universe (From Big-Bang !)

~ 14 billion years

= $14*10^9$ years

~ 10^{17} seconds

~ 10^{23} microseconds

An exponential time problem of moderate size, for example n = 100 would require more time than the age of the universe!

→ Computers are helpless to find exact solutions to most exponential time problems!