Data structures and algorithms Spring 2025

Analysis of Algorithms

Lecturer: Do Thuy Duong

Outline

- Algorithm analysis
- Time complexity
  - Experimental approach
  - Theoretical approach
    - Count primitive operations
- Worst case time complexity
- Big-Oh notation
- Rules for calculating Big-Oh
- Computational Complexity

Program performance

- Program performance is the amount of computer **memory/space** and **time** needed to run a program.
  - Memory/Space Complexity:
    - Amount of memory a program occupies
    - Usually measured in Bytes, KB or MB
  - Time Complexity:
    - Amount of time a program needs to run
    - Usually measured by the number of operations

Time complexity

- Time complexity is the amount of computer time a program needs to run.
- Why we care about time complexity?
  - Some computers require upper limits for program execution times.
  - Some programs require a real-time response.
  - Time is still a problem for us. If there are many solutions to a problem, we'll choose the fastest.
  - In some cases: Theoretically, we can solve the problem, but practically, we can't; it just takes too long when the input size gets larger. E.g: RSA public-key cryptosystems.

Measuring the Runtime (Experimental approach)

Calculate the actual running time:

- Write a program that implements the algorithm
- Run the program with data sets of varying size.
- Determine the **actual running time** using a system call to measure time (e.g. *System.currentTimeMillis()*);
- **Problem?**

Measuring the Runtime (Experimental approach)

Disadvantages:

1. Need to implement the algorithm, which may be difficult
2. Results are not indicative of the running time on other input size, which is not included in the experiment.
3. In order to compare two algorithms, the same hardware and software environments must be used.

Measuring the Runtime (Theoretical approach)

- Solution: Asymptotic Analysis.

→Evaluate the performance of an algorithm in terms of input size

→Doesn't measure the actual running time. Calculate how the time taken by an algorithm increases with the input size.

Measuring the Runtime (Theoretical approach)

Advantages:

- Use high-level description of the algorithms (**pseudo-code or diagrams**), instead of language dependent implementations.
- Measure the runtime as a function of the input size, n. (So, we know what happens when n gets larger)
- Care for all possible inputs.
- Measure the runtime independent of the hardware/software environment.

Measuring the Runtime (Theoretical approach)

- Pseudo-code: A description of an algorithm that is:
  - More structured than usual prose
  - Less formal than a programming language
- Example:

Algorithm arrayMax(A, *n*):
Input: An array A storing n integers.
Output: The maximum element in A.
Max ← A[0]
for *i*← **1 to** *n* **-1 do**
if *Max* < A[i] **then** *Max* ← A[i]
return *Max*

Measuring the Runtime (Theoretical approach)

- Control flow:
  - if … then … [else …]
  - while … do …
  - repeat … until …
  - for … do …
- Method declaration

  Algorithm name (param1, param2)

  Input …

  Output …

Measuring the Runtime (Theoretical approach)

- Method:
  - Calls: object method (args)
  - Returns: return value
- Expressions:

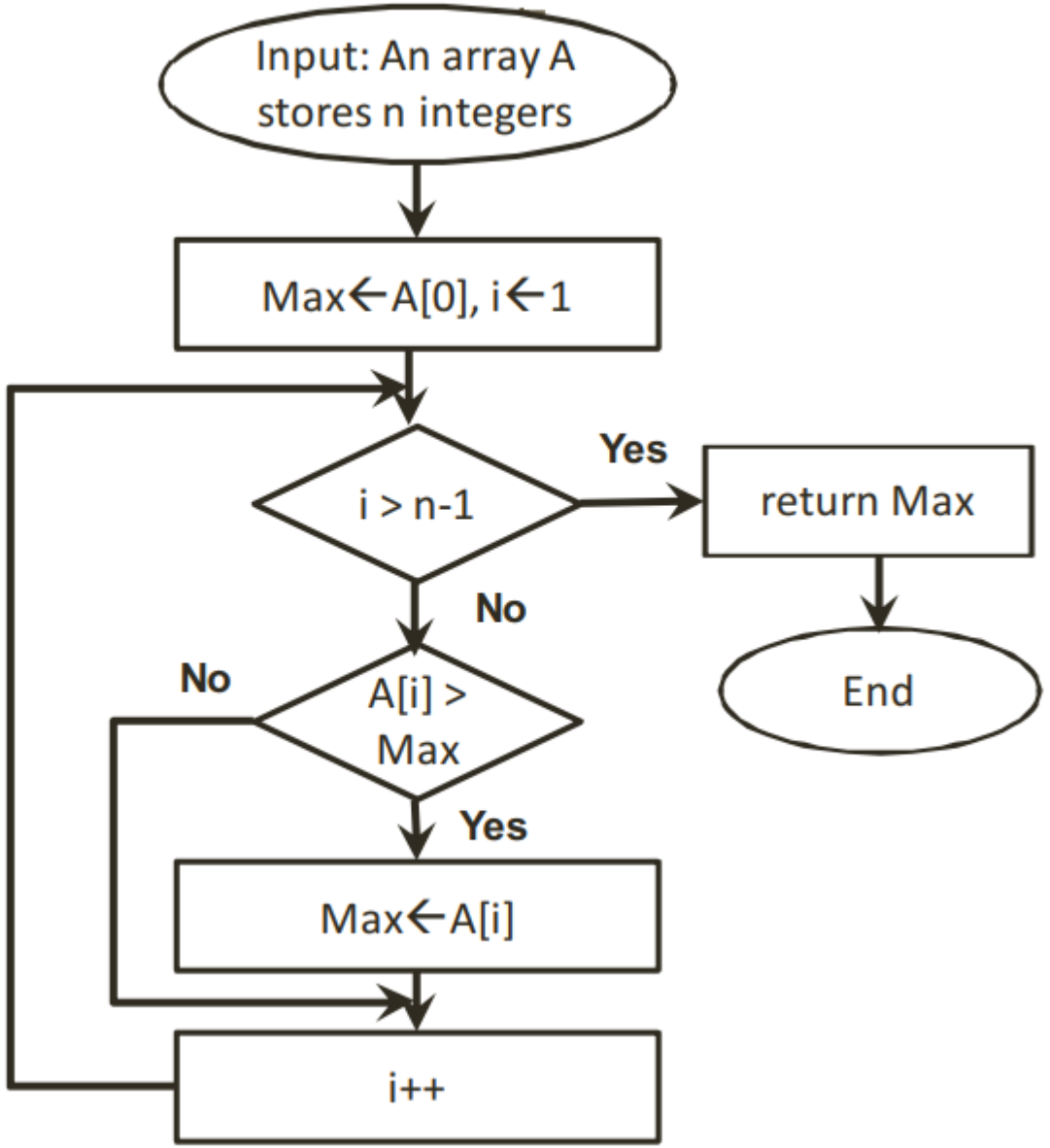
← Assignment (like = in Java)

= Equality testing (like == in Java)

n^2 Superscripts and other mathematical formatting allowed

Measuring the Runtime (Theoretical approach)

- Diagrams:



Theoretical approach
How to evaluate?

- Inspect the pseudo-code/diagram and **count the number of primitive operations** executed by the algorithm:
  - Make an addition, assignment = 1 operation
  - Calling a method or returning from a method = 1 operation.
  - Index in an array = 1 operation.
  - Comparison = 1 operation, etc.
- **T(n)**: a function of the input size n
- Primitive operations:
  - Basic computations performed by an algorithm
  - Largely independent from the programming language
  - Assumed to take a constant amount of time in the RAM model

Count primitive operations

- Inspect the pseudocode

→ count the maximum number of primitive operations executed by the algorithm as a function of the input size T(n)

- T(n) = 6n-1.

6n-1 operations

Code

Algorithm arrayMax(A, *n*):
Input: An array A storing n integers.
Output: The maximum element in A.

Max ← A[0] — 2 operations

2n operations (i=1 once, i<n n times, i++ n-1 times)

for *i*← **1 to** ***n*** **-1 do**

if *Max* < A[i] **then** — 2(n-1) operations

Max ← A[i] — 2(n-1) operations

return *Max* — 1 operation

Count primitive operations

- **T(n)** depends on the input data
- If A[0] is the maximum, then the code Max ← A[i] will never be executed (Best case):

T(n)=4n+1 (operations)

- If A[n-1] is the maximum, then the code Max ← A[i] will always be executed (Worst case):

T(n)=6n-1 (operations)

6n-1 operations

Code

Algorithm arrayMax(A, *n*):
Input: An array A storing n integers.
Output: The maximum element in A.

Max ← A[0] — 2 operations

for *i* ← **1 to** *n* **-1 do** — 2n operations (i=1 once, i<n n times, i++ n-1 times)
if *Max* < A[i] **then** — 2(n-1) operations
Max ← A[i] — 2(n-1) operations

return *Max* — 1 operation

Best case Time complexity

- The best case of an algorithm A is a function:
  $\mathbf{T_{BA}}: \mathbb{N} \rightarrow \mathbb{N}$ where $T_{BA}(n)$ is the minimum number of steps performed by A on an input of size n.

Worst case Time Complexity

The worst case of an algorithm A is a function:

$\mathbf{T_{WA}}: \mathbb{N} \rightarrow \mathbb{N}$ where $T_{WA}(n)$ is the maximum number of steps performed by A on an input of size n.

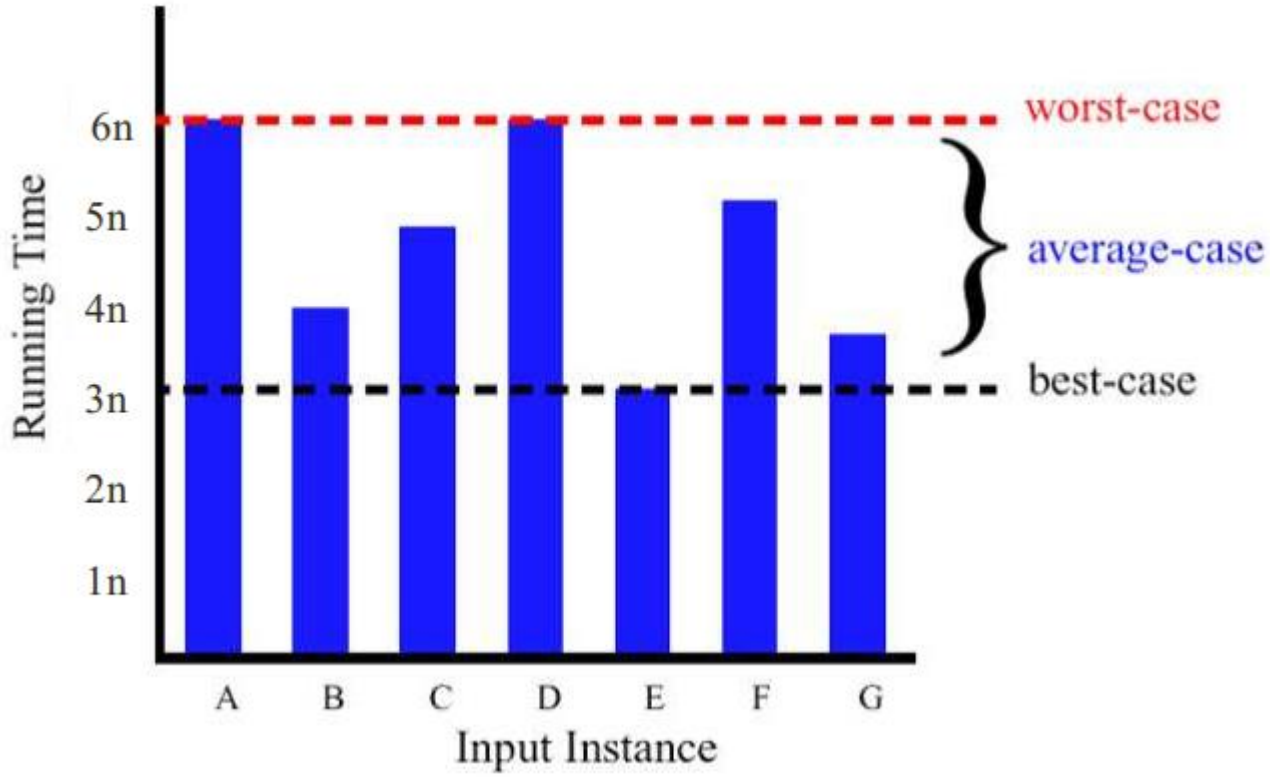
Average case time Complexity

- The ATC of an algorithm A is the function:

$T_{AVA}: \mathbb{N} \longrightarrow \mathbb{N}$ where $T_{AVA}(n)$ is the average number of steps performed by A on an input of size n.

- Problem with ATC:
  - What precisely does average mean? It depends on how the average input looks.
  - Average time analysis is mathematically very difficult and often infeasible

Worst case Time Complexity



Reasons for Worst-Case Analysis

Expect the best. Prepare for the worst → know in advance what happen when the input size gets huge (Programmers need to care when design algorithms).

- Many algorithms perform to their worst case a large part of the time.
- The best case is not very informative because many algorithms perform exactly the same in the best case → hard to compare their performance.
- Determining average-case performance is not always easy.
- The worst case gives us an **upper bound** on performance.

→ Use **worst-case running time as the main measure of time complexity**

Count primitive operations
More examples

```
public int powerTwo (int n)
{
        int result = n * n;
        return result;
}
```

Count primitive operations
More examples

```
public int powerTwo (int n)
{
        int result = n * n;
        return result;
}
```

T(n) = 3

Count primitive operations
More examples

```
public int search (int n, int array[])
{
        for (int i = 0, i<n, i++)
                if (array[i] == 0)
                        return 0;
        return 1;
}
```

Count primitive operations
More examples

```
public int search (int n, int array[])
{
        for (int i = 0, i<n, i++)
                if (array[i] == 0)
                        return 0;
        return 1;
}
```

T(n) = 1+(n+1)+3n+n+1 = 5n+3.

Count primitive operations
More examples

```
public void count (int n, int array[])
{
        int count = 0;
        for (int i = 0, i<n, i++)
                for (int j =0; j<n; j++)
                        if (array[i][j] == 0)
                                count += 1;
}
```

Count primitive operations
More examples

```
public void count (int n, int array[])
{
        int count = 0;
        for (int i = 0, i<n, i++)
                for (int j =0; j<n; j++)
                        if (array[i][j] == 0)
                                count += 1;
}
```

$T(n) = 1+\{1+(n+1)+n*[1+(n+1)+5n+n]+n\} = 2n+3+n*[7n+2] = 2n+3+7n^2+2n$

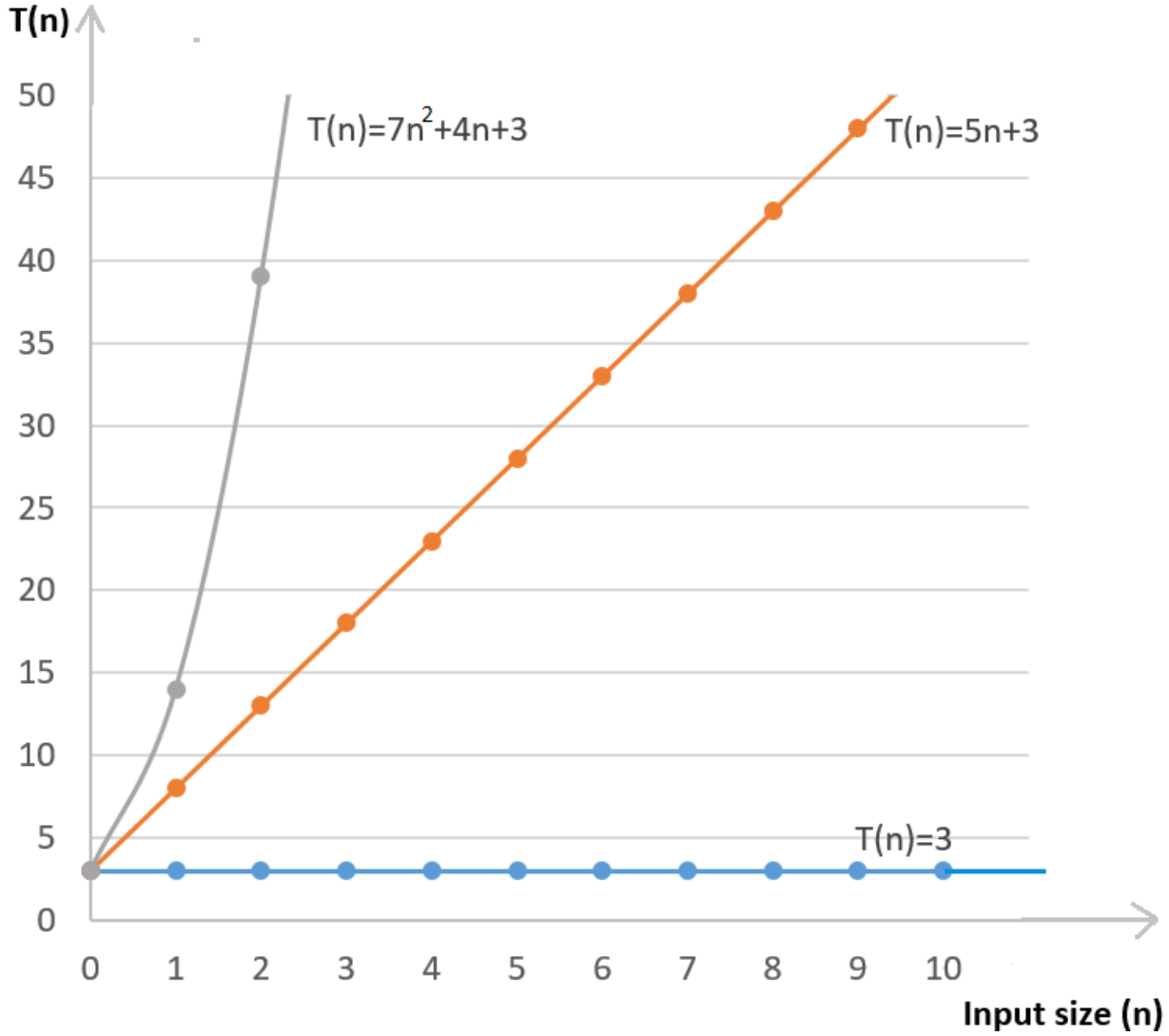
$= 7n^2+4n+3.$

Count primitive operations

When n gets larger,

which of the three programs

runs the fastest?



Count primitive operations

Consider T(n):

$T(n)=2^n+3n+5$ $T(n)=n^3+3n^2-4n+8$

$T(n)=c_k n^k+c_{k-1}n^{k-1}+c_{k-2}n^{k-2}\ldots+c_1 n+c_0$

$T(n)=c_n 2^n+c_{n-1}2^{n-1}+c_{n-2}2^{n-2}\ldots+c_1 2^1+c_0$

- Too complicated
- Too many terms
- Difficult to compare two expressions, each with 10 or 20 terms

Do we really need that many terms?

Asymptotic behavior

- Consider: $T(n) = n^3 + 10n^2 + 20n + 500$

Note:

- When $n = 1{,}000$

$T(n) = \mathbf{1{,}000{,}000{,}000} + 10{,}000{,}000 + 20{,}000 + 500$

- The value (behavior) of $T(n)$ when n gets large is determined entirely by the leading term ($\mathbf{n^3}$)
- We may drop all except the $\mathbf{n^3}$ term

→ **The leading term (fastest growing term) will dominate the running time**

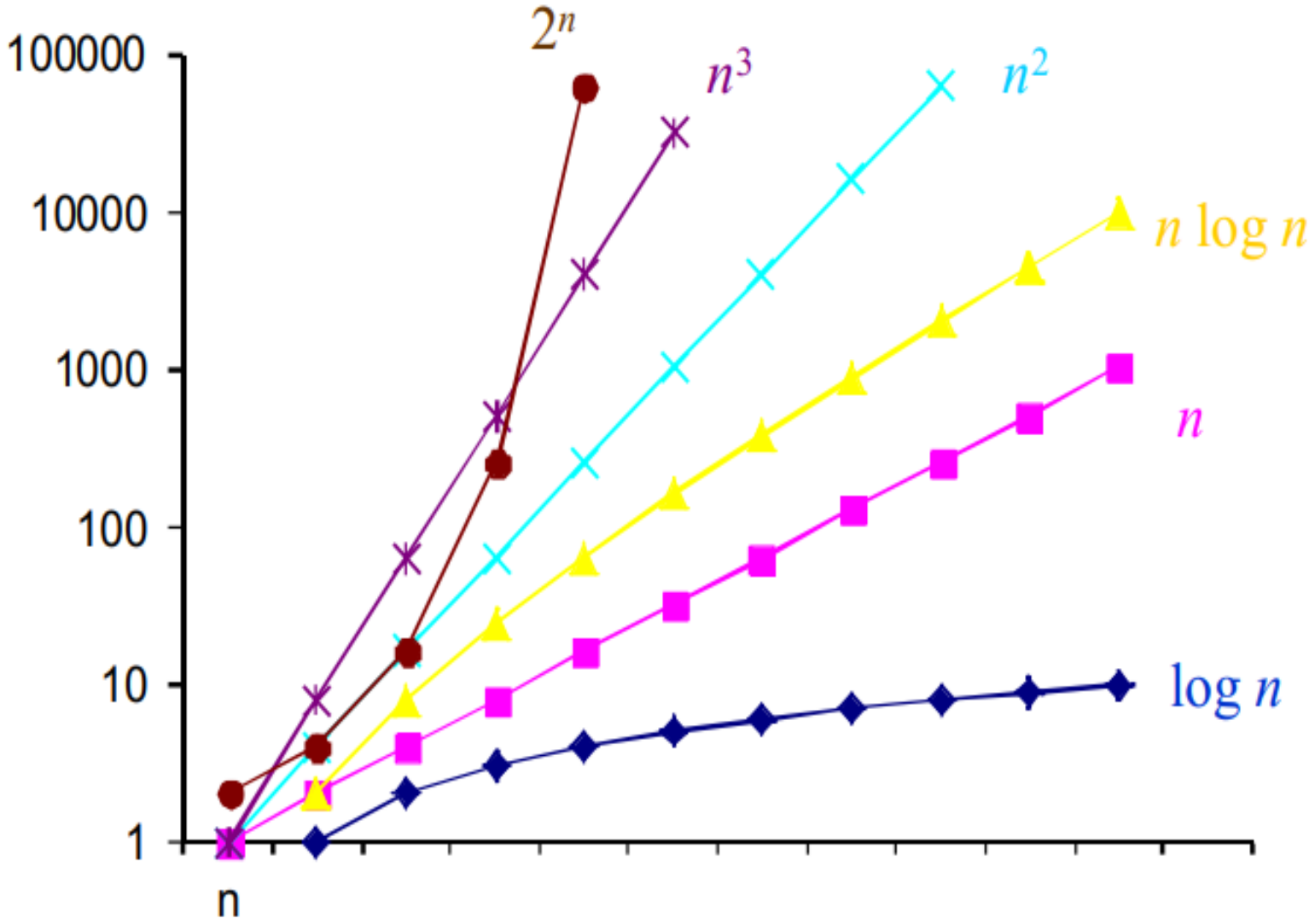
Asymptotic behavior

To compare the time complexity of two algorithm we **compare the growing rate** of the leading terms. (Note: compare n^3 and 2^n when n = 1, 2, 4, 8,…)

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1,024	32,768	4,294,967,296

Asymptotic behavior

To compare
the time complexity
of algorithms, we
compare the growing rate
of the leading terms.



- We only need to know the leading term of the T(n) to evaluate time complexity of an algorithm.

→ Calculate Big – Oh of the algorithm?

Big-Oh Notation

- Big-Oh represents the worst case time complexity of an algorithm.

Big-Oh Notation

The Big-Oh notation definition:

• Let f, g: $\mathbb{N} \rightarrow R$ be functions.

f(n) is O(g(n)) if there is an $n_0 \in N$ and a $c > 0$ in R such that

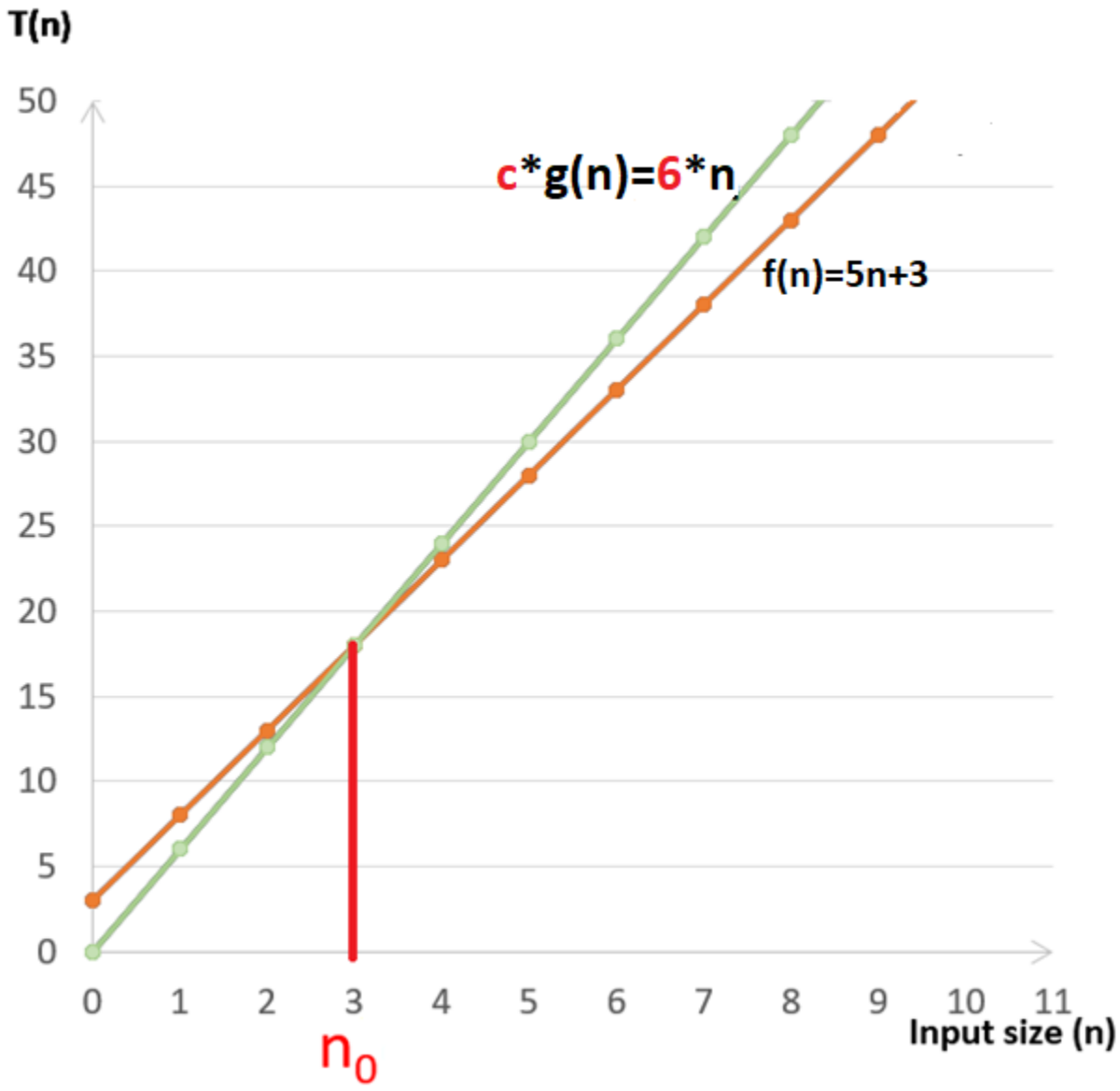
$\forall n \geq n_0$ we have $f(n) \leq c*g(n)$.

• Example:

$T(n)=f(n)=5n+3 \leq 6*n \ \forall n \geq 3$

$T(n)=f(n) \leq 6*n \ \forall n \geq 3$

→ $g(n) = n$ → $f(n)$ is $O(n)$.



The Big-Oh notation definition:

• Let f, g: $\mathbb{N} \rightarrow R$ be functions.

f(n) is O(g(n)) if there is an $n_0 \in N$ and a $c > 0$ in R such that

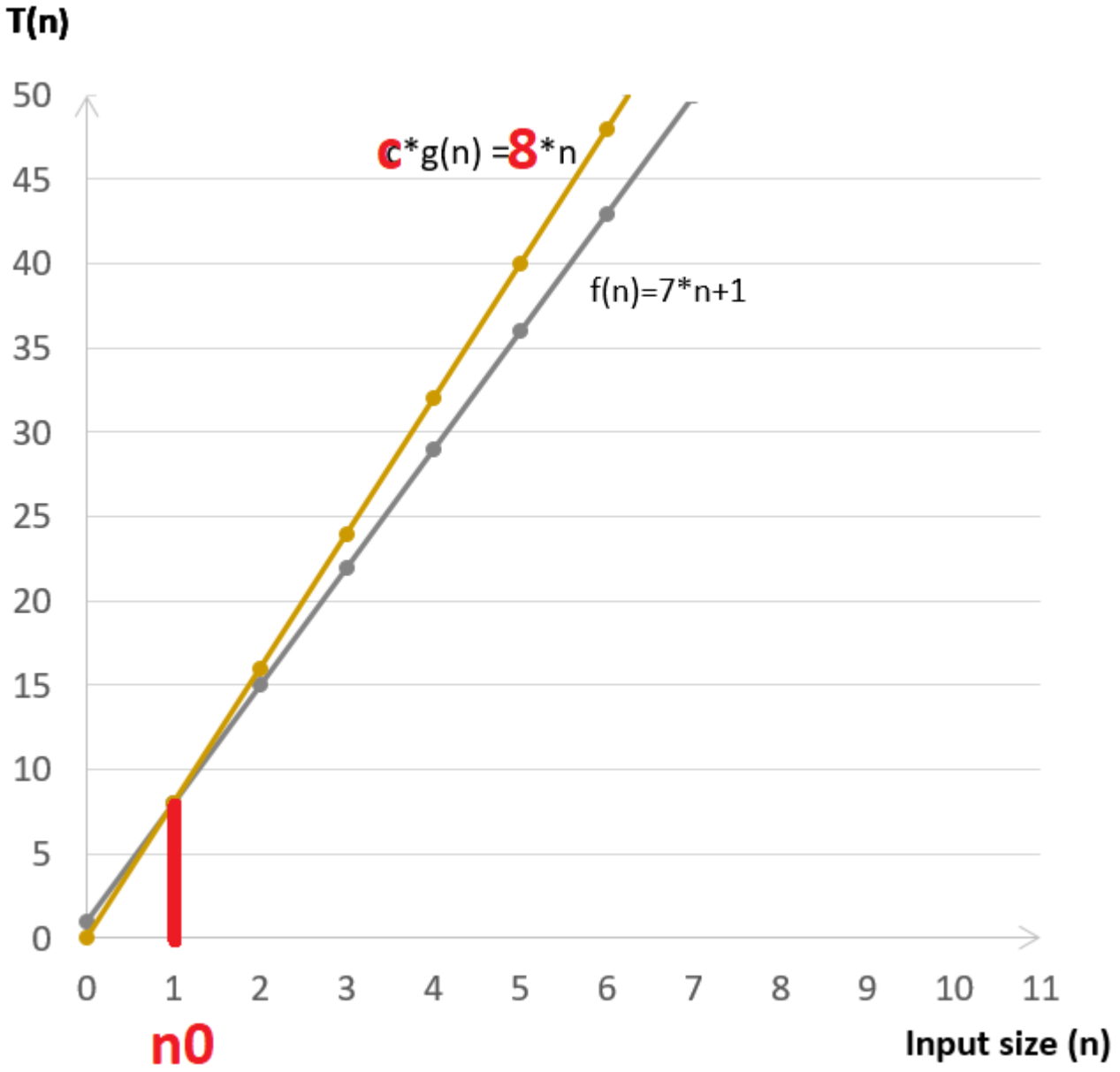
$\forall n \geq n_0$ we have $f(n) \leq c*g(n)$.

- Example:

$f(n)=7n+1 \leq 8*n \ \forall n \geq 1$

$f(n) \leq 8*n \ \forall n \geq 1$

→ g(n) = n → f(n) is O(n).



Big-Oh notation

Big-Oh represents an **upper-bound** of the **growth rate of a function**.

- If an algorithm needs **T(n) = 5n+3** basic operations and another needs **T(n) = 7n+1** basic operations, we will consider them to be in the ***same efficiency category***. (they have the **same Big-Oh notation O(n)**)

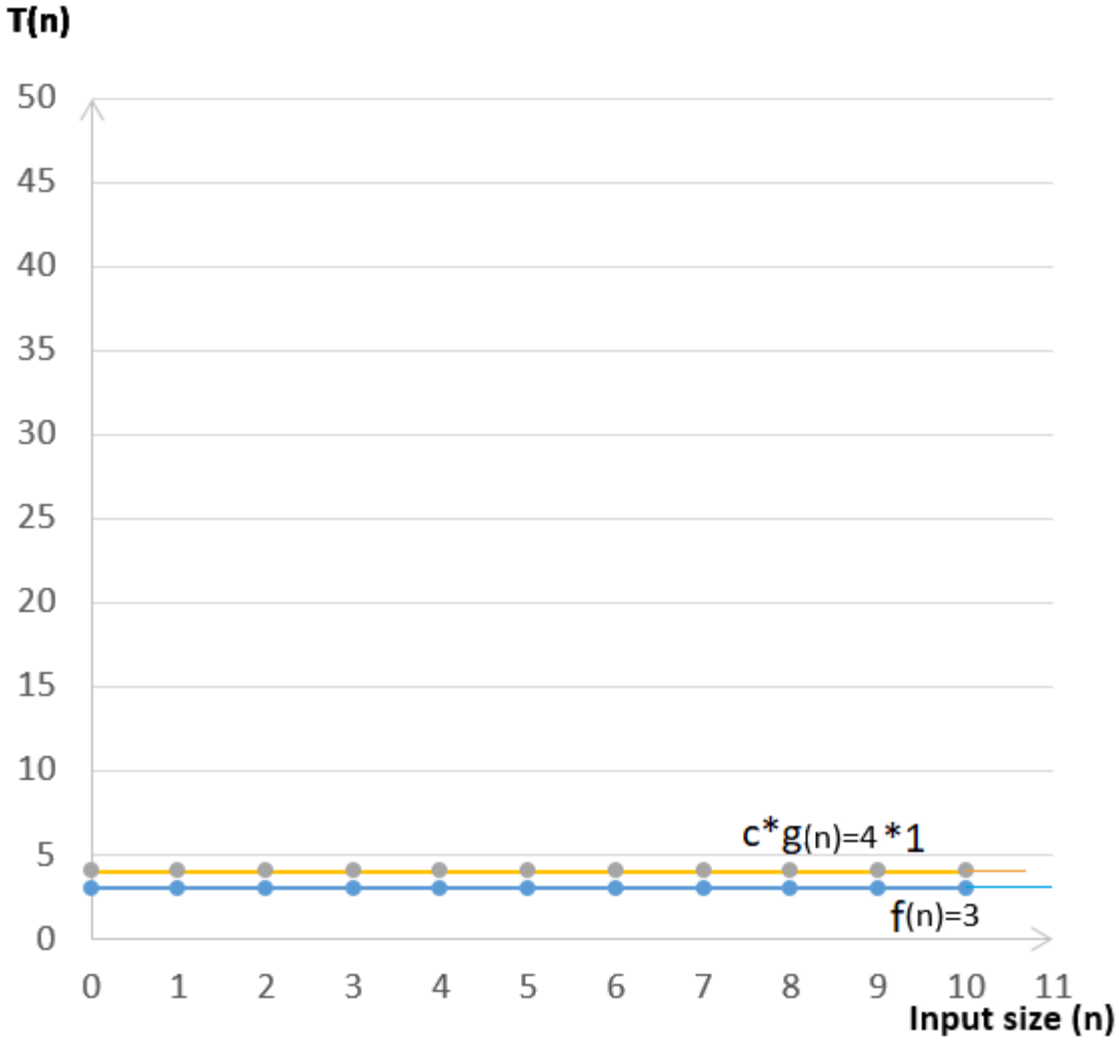
Big-Oh Notation

- Example:

$f(n)=3 \leq 4*1 \ \forall n \geq 0$

$f(n) \leq 4*1 \ \forall n \geq 0$

→ $g(n) = 1$ → $f(n)$ is $O(1)$.



Big-Oh Notation

- Big-Oh notation proof:
  - Let f(n) = K (K is a constant), prove that f(n) is O(1)

  (which means g(n) = 1)
  - Solution: We must find n_0 and c such that for all $n \geq n_0$, we have f(n) ≤ c*g(n)

  Choose c = K+1 and $n_0 = 1$

  →for all $n \geq 1$: f(n) = K<(K+1)* (1) → g(n) = 1 → f(n) = O(1).

Big-Oh Notation

- Big-Oh notation proof:
  - Let $f(n) = 2n^2 + K$ (K is a constant), prove that f(n) is $O(n^2)$

  (which means $g(n) = n^2$)

  - Solution: We must find n_0 and c such that for all $n \geq n_0$, we have $f(n) \leq c*g(n)$

  Prove: $2n^2 + K \leq c*n^2 = (c-1)*n^2 + n^2$

  We choose $c = 3$ and $n_0 \geq \sqrt{K}$ → $2n^2 + (\sqrt{K})^2 \leq (3-1)*n^2 + n^2$ → f(n) is O(g(n)) $= O(n^2)$

Big-Oh notation

- More Big-Oh notation example:

- $\frac{n^2}{2} + 3n + 1$ is $O(n^2)$
- $7n^3 + 10n^2 + 3$ is $O(n^3)$
- $2^n + n^3 + n^2 + 1$ is $O(2^n)$
- $\log_{10} n$ is $O(\log n)$
- $\sin(n)$ is $O(1)$, 10 is $O(1)$, 10^{10} is $O(1)$

	$O(g(n))$						
n	n	$n \log n$	n^2	n^3	n^4	n^{10}	2^n
10	.01µs	.03µs	.1µs	1µs	10µs	10s	1µs
20	.02µs	.09µs	.4µs	8µs	160µs	2.84h	1ms
30	.03µs	.15µs	.9µs	27µs	810µs	6.83d	1s
40	.04µs	.21µs	1.6µs	64µs	2.56ms	121d	18m
50	.05µs	.28µs	2.5µs	125µs	6.25ms	3.1y	13d
100	.1µs	.66µs	10µs	1ms	100ms	3171y	4×10^{13}y
10^3	1µs	9.96µs	1ms	1s	16.67m	3.17×10^{13}y	32×10^{283}y
10^4	10µs	130µs	100ms	16.67m	115.7d	3.17×10^{23}y	
10^5	100µs	1.66ms	10s	11.57d	3171y	3.17×10^{33}y	
10^6	1ms	19.92ms	16.67m	31.71y	3.17×10^{7}y	3.17×10^{43}y	

Estimate Big-Oh of an algorithm

- Method 1
  - ✓ Analyze the algorithm, estimate the total number of primitive operations **T(n)**.
  - ✓ Find T(n) is O(g(n)).
- Method 2
  - ✓ Analyze the algorithm, estimate the Big-Oh for each blocks

Method 1

Code

Hard to find T(n)

```
public static int FindMaxSubSeqSum1(int [] a)
    {
        int maxSum=0;
        for (int i=0; i<a.length; i++)
            for (int j=i; j<a.length; j++)
            {
                int thisSum=0;
                for (int k=i; k<=j; k++)
                    thisSum+=a[k];
                if (thisSum>maxSum)
                    maxSum=thisSum;
            }
        return maxSum;
    }
```

Rules for calculating Big-Oh

Let $d, e, f, g : \mathbb{N} \to \mathbb{R}$ *be functions. Then:*

(1) For any constant $a > 0$ *in* $\mathbb{R}$*:*

$$d(n) \in O\big(f(n)\big) \implies a \cdot d(n) \in O\big(f(n)\big).$$

(2) $d(n) \in O\big(f(n)\big)$ *and* $e(n) \in O\big(g(n)\big) \Rightarrow d(n) + e(n) \in O\big(f(n) + g(n)\big)$

(3) $d(n) \in O\big(f(n)\big)$ *and* $e(n) \in O\big(g(n)\big) \Rightarrow d(n) \cdot e(n) \in O\big(f(n) \cdot g(n)\big).$

Rules for calculating Big-Oh

(5) *For any* $d \in \mathbb{N}$: $f(n)$ *polynomial of degree* $d \implies f(n) \in O(n^d)$.

(6) *For any* $x > 0, a > 1$ *in* $\mathbb{R}$: $n^x \in O(a^n)$.

(7) *For any* $x > 0$ *in* $\mathbb{R}$: $\lg(n^x) \in O\big(\lg(n)\big)$.

(8) *For any* $x > 0, y > 0$ *in* $\mathbb{R}$: $\lg^x(n) \in O(n^y)$.

Rules for calculating Big-Oh

- **Rule for sums:**

  $O(f(n)) + O(g(n))$ *is* $O(max(f(n), g(n)))$

- **Rule for products**

  $O(f(n)).O(g(n))$ *is* $O(f(n).g(n))$

- The running time of each assignment, read, write statement can be taken to be O(1)
- The running time of a sequence of statements is determined by the sum rule
- The running time of an if-statement is the cost of the conditionally executed statements, plus the time for evaluating the condition..
- The time to execute a loop is the sum, over all times around the loop.

Method 2 – General rules

- Algorithm contains several blocks

Code

```
Block A
Block B
Block C
```

$$O() = O_A() + O_B() + O_C()$$
$$= \max\{O_A(), O_B(), O_C()\}$$

- Algorithm contains statements independent to the input size n

Code

```
a=0; c=d+e; …
for (int i=0; i<1000; i++)
{
  x=y+z; b++; …
}
return s;
```

$$O(1)$$

Method 2 – General rules

- Conditional block

Code

```
if (condition)
{
        Block A;
} else
{
        Block B;
}
```

$O()=\max\{O_A(), O_B()\}$

- Loop block

Code

```
for (int i=0; i<n; i++)
{
        Block A
}
```

$O()=O(n).O_A()$

Method 2 – General rules

- Nested loop:

Code

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

$O() = O(n).O(n).O_A() = O(n^2).\ O_A()$

- Function call:

Code

```
s=funcA();
```

$O() = O_A()$

Example

$O(1)+ O(n^3) +O(1)= O(n^3)$

Code

```
public static int FindMaxSubSeqSum1(int [] a)
  {
    int maxSum=0;                                   O(1)

        for (int i=0; i<a.length; i++)              O(n).O(n^2) = O(n^3)

            for (int j=i; j<a.length; j++)          O(n).O(n) = O(n^2)
            {                                       O(1)
                int thisSum=0;                      O(1)+O(n)+O(1) = O(n)

                for (int k=i; k<=j; k++)            O(n).O(1)=O(n)
                    thisSum+=a[k];                  O(1)

                if (thisSum>maxSum)                 O(1)
                    maxSum=thisSum;
            }

    return maxSum;                                  O(1)
  }
```

Exa

- Finding maximum value in an array?

```
for (i = 0; i < a.length; i++) {
   if (max < a[i] ) {
         max = a[i];
   }
}
```

- Summing a list of numbers

```
for (i = 0; i < a.length; i++) {
   sum += a[i];
}
```

Example 3

```
for (i = 0; i < N; i++)
   for (j = 0; j < N * N; j++)
      sum++;

for (i = 0; i < N; i++)
   for (j = 0; j < i; j++)
      sum++;

for (i = 1; i < N; i++)
   for(j = 1; j < i * i; j++)
       sum++

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

Recurrence relation

- Consider the recursive method factorial:

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

Recurrence relation

- The runtime of the program

$$\begin{cases} T(n) = 1 \text{ if } n = 0 \\ T(n) = T(n-1) + 1 \text{ if } n > 0 \end{cases}$$

- **Method 1: Substitution method**

$T(n) = T(n-1) + 1$

$= (T(n-2) + 1) + 1 = T(n-2) + 2$

$= T(n-3) + 3$

$= \ldots$

$= T(n-n) + n$

$= T(0) + n = n+1$

→ $T(n)$ is $O(n)$

Recurrence relation

```
public void Test (int n)
{
    if (n>0)
    {
        for (int i=0; i<n;i++)
            System.out.println(i);
    }
    Test(n-1);
}
```

Recurrence relation

- Method 2: Recurrence Tree Method
- Method 3: Master Theorem Method

Tutorial and next topic

Preparing for the tutorial:

- Practice with examples and exercises in Tutorial 2
- Read textbook section 1.3 (Recursion), reference book chapter 3

Preparing for next topic:

- Read textbook chapter 7 Sorting (7.2 & 7.6)
- Reference book chapter 4
  - Divide and Conquer
  - Evaluate T(n) of recursion functions