

Algorithm Efficiency Analysis

Nor Bahiah Hj Ahmad & Dayang Norhayati A.Jawawi
School of Computing

Objectives

At the end of the class, students are expected to be able to do the following:

- Know how to measure algorithm efficiency.
- Know the meaning of big O notation and determine the notation in algorithm analysis.

Algorithm analysis

- Study the efficiency of algorithms when the input size grow based on the number of steps, the amount of computer time and space.
- Is a major field that provides tools for evaluating the efficiency of different solutions.
- What is an efficient algorithm?
 - ✓ Faster is better (Time) - How do you measure time? Wall clock? Computer clock?
 - ✓ Less space demanding is better - But if you need to get data out of main memory it takes time

Analysis of algorithms

Algorithm analysis should be independent of :

- Specific implementations and coding tricks (programming language, control statements - Pascal, C, C++, Java)
- Specific Computers (hardware chip, OS, clock speed)
- Particular sets of data (string, int, float).

But size of data should matter

Analysis of algorithms

For a particular problem size, we may be interested in:

- **Worst-case efficiency:** Longest running time for any input of size n

A determination of the maximum amount of time that an algorithm requires to solve problems of size n

- **Best-case efficiency:** Shortest running time for any input of size n

A determination of the minimum amount of time that an algorithm requires to solve problems of size n

- **Average-case efficiency:** Average running time for all inputs of size n

A determination of the average amount of time that an algorithm requires to solve problems of size n .

Complexity of algorithm

- Complexity time can be represented by **big 'O' notation**
- Big 'O' notation is denoted as: **$O(acc)$**
whereby:
 - ✓ **O** - order
 - ✓ **acc** - class of algorithm complexity
- Big O notation example:
 $O(1)$, $O(\log_x n)$, $O(n)$, $O(n \log_x n)$, $O(n^2)$

Big O notation

Notation	Execution time/ number of step
$O(1)$	Constant function, independent of input size, n . Example: Finding the first element of a list.
$O(\log_x n)$	Problem complexity increases slowly as the problem size increases. Squaring the problem size only doubles the time. Characteristic: Solve a problem by splitting into constant fractions of the problem (e.g., throw away $\frac{1}{2}$ at each step)
$O(n)$	Problem complexity increases linearly with the size of the input, n Example: counting the elements in a list.

Big O notation

Notation	Execution time/ number of step
$O(n \log n)$	<p>Log-linear increase - Problem complexity increases a little faster than n</p> <p>Characteristic: Divide problem into sub problems that are solved the same way.</p> <p>Example: Merge sort</p>
$O(n^2)$	<p>Quadratic increase.</p> <p>Problem complexity increases fairly fast, but still manageable</p> <p>Characteristic: Two nested loops of size n</p>
$O(n^3)$	<p>Cubic increase.</p> <p>Practical for small input size, n.</p>
$O(2^n)$	<p>Exponential increase - Increase too rapidly to be practical</p> <p>Problem complexity increases very fast</p> <p>Generally unmanageable for any meaningful n</p> <p>Example: Find all subsets of a set of n elements</p>

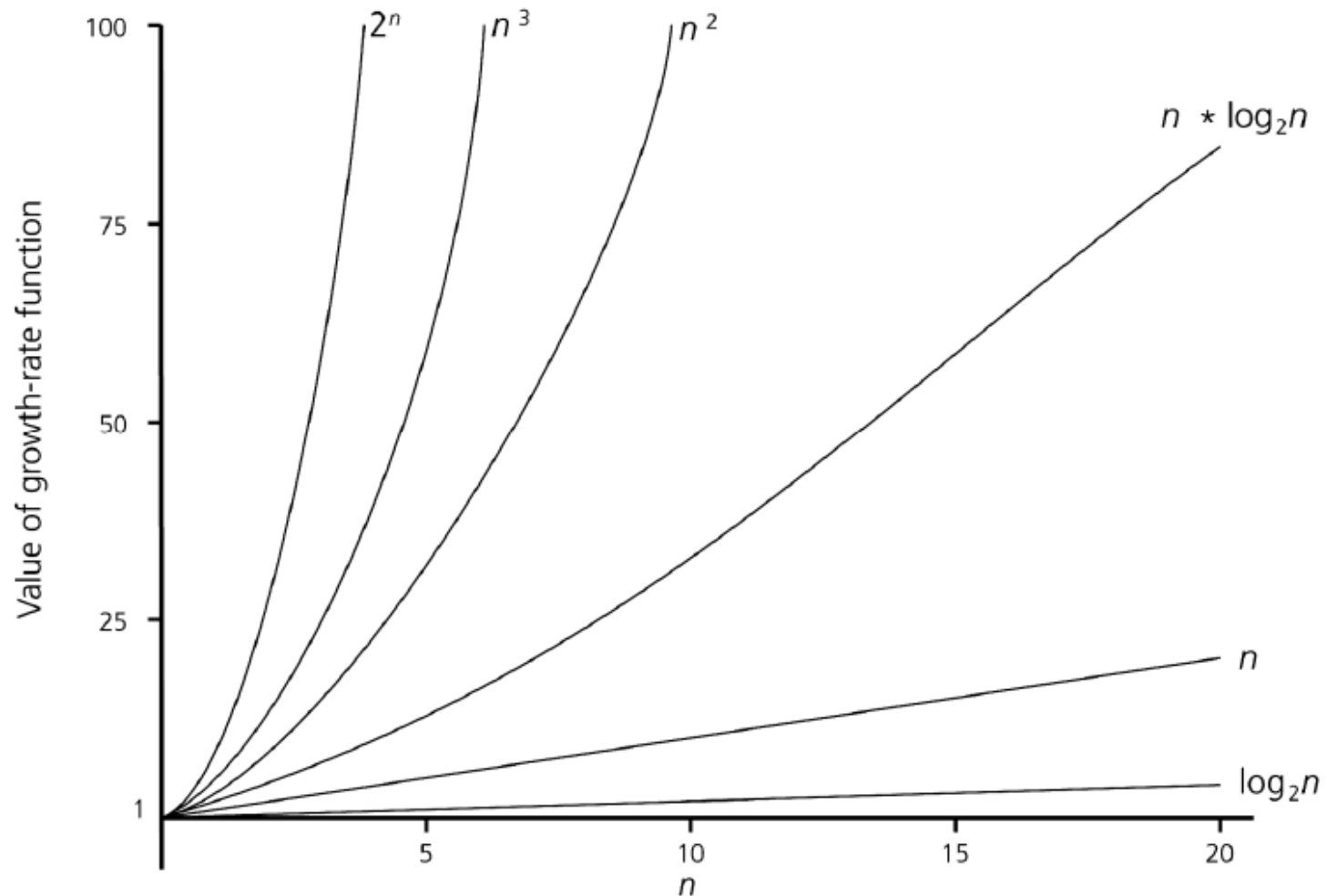
Order-of-Magnitude Analysis and Big O Notation

A comparison of growth-rate functions in tabular form

Function	n					
	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Order-of-Magnitude Analysis and Big O Notation

A comparison of growth-rate functions in graphical form



Order of increasing complexity

$$O(1) < O(\log_x n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n)$$

Notasi	n = 8	n = 16	n = 32
$O(\log_2 n)$	3	4	5
$O(n)$	8	16	32
$O(n \log_2 n)$	24	64	160
$O(n^2)$	64	256	1024
$O(n^3)$	512	4096	32768
$O(2^n)$	256	65536	4294967296

Big O Notation

Example of algorithm (only for cout operation):

Notation	Code
$O(1)$	<pre>int counter = 1; cout << "Arahan cout kali ke " << counter << "\n";</pre>
$O(\log_x n)$	<pre>int counter = 1; int i; for (i = x; i <= n; i = i * x) { // x must be > than 1 cout << "Arahan cout kali ke " << counter << "\n"; counter++; }</pre>
$O(n)$	<pre>int counter = 1; int i; for (i = 1; i <= n; i++) { cout << "Arahan cout kali ke " << counter << "\n"; counter++; }</pre>

Big O Notation

Example of algorithm (only for cout operation):

Notation	Code
$O(n \log_x n)$	<pre>int counter = 1; int i, j = 1; for (i = x; i <= n; i = i * x) { // x must be > than 1 while (j <= n) { cout << "Arahan cout kali ke " << counter << "\n"; counter++; j++; } }</pre>

Big O Notation

Example of algorithm (only for cout operation):

Notation	Code
$O(n^2)$	<pre>int counter = 1; int i, j; for (i = 1; i <= n; i++) { for (j = 1; j <= n; j++) { cout << "Arahan cout kali ke " << counter << "\n"; counter++; } }</pre>

Big O Notation

Example of algorithm (only for cout operation):

Notation	Code
$O(n^3)$	<pre>int counter = 1; int i, j, k; for (i = 1; i <= n; i++) { for (j = 1; j <= n; j++) { for (k = 1; k <= n; k++) { cout << "Arahan cout kali ke " << counter << "\n"; counter++; } } }</pre>

Big O Notation

Example of algorithm (only for cout operation):

Notation	Code
$O(2^n)$	<pre>int counter = 1; int i = 1, j = 1; while (i <= n) { j = j * 2; i++; } for (i = 1; i <= j; i++) { cout << "Arahan cout kali ke " << counter << "\n"; counter++; }</pre>

Determine the complexity time of algorithm

Can be determined

- Theoretically - by calculation

The complexity time is related to the number of steps/operations

- ✓ Count the number of steps and then find the class of complexity @
- ✓ Find the complexity time for each steps and then count the total

- Practically – by experiment or implementation

- ✓ Implement the algorithms in any programming language and run the programs
- ✓ Depend on the compiler, computer, data input and programming style.

Determine the number of steps

It can be expressed by **summation series**

$$\sum_{i=1}^n f(i) = f(1) + f(2) + \dots + f(n) = n$$

where:

$f(i)$ – Statement executed in the loop

Example 1: If $n = 5, i = 1$

$$\sum_{i=1}^5 f(i) = f(1) + f(2) + f(3) + f(4) + f(5) = 5$$

The statement that represented by $f(i)$ will be **repeated 5 times**

Determine the number of steps

Example 2: If $n = 5, i = 3$

$$\sum_{i=3}^5 f(i) = f(3) + f(4) + f(5) = 3$$

The statement that represented by $f(i)$ will be repeated **3 times**

Example 3: If $n = 1, i = 1$

$$\sum_{i=1}^1 f(i) = f(1) = 1$$

The statement that represented by $f(i)$ will be executed **only once**

Determine the number of steps

Example 1

Statements	Number of steps
int counter = 1;	$\sum_{i=1}^1 f(i) = 1$
int i = 0;	$\sum_{i=1}^1 f(i) = 1$
for (i = 1; i <= n; i++) {	$\sum_{i=1}^n f(i) = n$
cout << "Arahan cout kali ke " << counter << "\n";	$\sum_{i=1}^n f(i) \sum_{i=1}^1 f(i) = n.1 = n$
counter++;	$\sum_{i=1}^n f(i) \sum_{i=1}^1 f(i) = n.1 = n$
}	0
Total Steps	2 + 3n
Complexity Time	O(n)

Determine the number of steps

Example 2

Algorithm	Number of steps
void sample4 () {	0
for (int a=2; a<=n; a++)	$n - 2 + 1 = n - 1$
cout << "Example of step calculation";	$(n - 1).1 = n - 1$
}	0
Total Steps	$2(n - 1)$
Complexity Time	$O(n)$

Determine the number of steps

Example 3

Algorithm	Number of steps
void sample5 () {	0
for (int a=1; a<=n-1; a++)	$n - 1 - 1 + 1 = \mathbf{n - 1}$
cout << "Example of step calculation";	$\mathbf{(n - 1).1 = n - 1}$
}	0
Total Steps	$\mathbf{2(n - 1)}$
Complexity Time	$\mathbf{O(n)}$

Determine the number of steps

Example 4

Algorithm	Number of steps
void sample6 () {	0
for (int a=1; a<=n; a++)	$n - 1 + 1 = n$
for (int b=1; b<=n; b++)	$n \cdot (n - 1 + 1) = n \cdot n$
cout << "Example of step calculation";	$n \cdot n \cdot 1 = n \cdot n$
}	0
Total Steps	$n + 2n^2$
Complexity Time	$O(n^2)$

Determine the number of steps

Example 5

Algorithm	Number of steps
void sample6 () {	0
for (int a=1; a<=n; a++)	$n - 1 + 1 = \mathbf{n}$
for (int b=1; b<=a; b++)	$\mathbf{n(n + 1) / 2}$
cout << "Example of step calculation";	$\mathbf{(n(n + 1) / 2).1 = n(n+1)/2}$
}	0
Total Steps	$\mathbf{n + n^2 + n = 2n + \mathbf{n^2}}$
Complexity Time	$\mathbf{O(n^2)}$

Determine the number of steps

To get **$n.(n+1)/2$** , we used summation series as shown below:

$$\begin{aligned}\sum_{a=1}^n \sum_{b=1}^n 1 &= \\ &= n(1 + 2 + 3 + 4 + \dots + n) \\ &= \frac{n(n+1)}{2} \\ &= \frac{n^2 + n}{2}\end{aligned}$$

Proof:

If **$n = 4$** , for inner loop
`for (int b=1; b<=a; b++)`

It will be repeated:
 $(1+2+3+4) = 10$ times

By using the formula:
 $n.(n+1)/2 = 4(5)/2 = 10$ times

Determine the number of steps

Example 6

Statements	Number of steps
int counter = 1;	$\sum_{i=1}^1 f(i) = 1$
int i = 0;	$\sum_{i=1}^1 f(i) = 1$
int j = 1;	$\sum_{i=1}^1 f(i) = 1$
for (i = 3; i <= n; i = i * 3) {	$\sum_{i=3}^n f(i) = f(3) + f(9) + f(27) + \dots + f(n) = \log_3 n$
while (j <= n) {	$\sum_{i=3}^n f(i) \sum_{i=1}^n f(i) = n \cdot \log_3 n$
cout << "Arahan cout kali ke " << counter << "\n";	$\sum_{i=3}^n f(i) \sum_{i=1}^n f(i) \sum_{i=1}^1 f(i) = n \cdot \log_3 n \cdot 1$
counter++;	$\sum_{i=3}^n f(i) \sum_{i=1}^n f(i) \sum_{i=1}^1 f(i) = n \cdot \log_3 n \cdot 1$
j++;	$\sum_{i=3}^n f(i) \sum_{i=1}^n f(i) \sum_{i=1}^1 f(i) = n \cdot \log_3 n \cdot 1$
}}	0
Total Steps	$3 + \log_3 n + 4n \log_3 n$
Complexity Time	$O(n \log_2 n)$

Determine the number of steps

$$3 + \log_3 n + 4n \log_3 n$$

- Consider the largest factor: $4n \log_3 n$
- Remove the coefficient: $n \log_3 n$
- In **asymptotic classification**, the base of the log can be omitted as shown in this formula:

$$\log_a n = \log_b n / \log_b a$$

- Then,

$$\log_3 n = \log_2 n / \log_2 3 = \log_2 n / 1.58$$

- Remove the coefficient $1/1.58$
- So, the **complexity time** = $O(n \log_2 n)$

Determine the number of steps

Example 7

Algorithm	Number of steps
void sample8 () {	0
int n, x, i=1;	1
while (i<=n) {	$n - 1 + 1 = n$
x++;	$n.1 = n$
i++;	$n.1 = n$
}	0
Total Steps	$1 + 3n$
Complexity Time	$O(n)$

Determine the number of steps

Example 8

Algorithm	Number of steps
void sample9 () {	0
int n, x, i=1;	1
while (i<=n) {	$1 + \log_2 n$
x++;	$(1 + \log_2 n) \cdot 1 = 1 + \log_2 n$
i=i*2;	$(1 + \log_2 n) \cdot 1 = 1 + \log_2 n$
}	0
Total Steps	$1 + 3(1 + \log_2 n)$
Complexity Time	$O(\log_2 n)$

Determine the number of steps

Example 9

Algorithm	Number of steps
void sample9 () {	0
int n, x, i=1;	1
while (i<=n) {	$1 + \log_4 n$
x++;	$(1 + \log_4 n) \cdot 1 = 1 + \log_4 n$
i=i*4;	$(1 + \log_4 n) \cdot 1 = 1 + \log_4 n$
}	0
Total Steps	$1 + 3(1 + \log_4 n)$
Complexity Time	$O(\log_2 n)$

- While loop iterate from $i=1$ until $i=n$; i increment 4 times at each iteration
- Number of iteration for while loop = **$(1 + \log_4 n)$**

Determine the number of steps

Using **asymptotic O**, where base for the algorithm is ignored:

$$\log_a n = \log_b n / \log_b a$$

Then,

$$\log_4 n = \log_2 n / \log_2 4 = \log_2 n / 2$$

Therefore,

$$\text{Complexity Time} = O(\log_2 n)$$

Thank You