# The Complexity of Algorithms (3A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

# Complexity Analysis

- to compare algorithms at the idea level ignoring the low level details
- To measure how fast a program is
- To explain how an algorithm behaves as the input grows larger

https://discrete.gr/complexity/

# Counting Instructions

| | |
|---|---|
| • Assigning a value to a variable | x= 100; |
| • Accessing a value of a particular array element | A[i] |
| • Comparing two values | (x > y) |
| • Incrementing a value | i++ |
| • Basic arithmetic operations | +, –, *, / |
| • Branching is not counted | if else |

https://discrete.gr/complexity/

# Asymptotic Behavior

- avoiding tedious instruction counting
- eliminate all the minor details
- focusing how algorithms behaves when treated badly
- drop all the terms that grow slowly
- only keep the ones that grow fast as **n** becomes larger

https://discrete.gr/complexity/

# Finding the Maximum

```
M = A[0];
for (i=0; i<n; ++i) {
    if (A[i] >= M) {
        M = A[i];
    }
}
```

// M is set to the 1st element

// if the (i+1)th element is greater than M,

// M is set to that element (new maximum value)

```
int A[n];    // n element integer array A
int M;       // the current maximum value found so far
             // set to the 1st element, initially
```

https://discrete.gr/complexity/

# Worst and Best Cases

int A[4];

| | |
|---|---|
| i=0 | A[0] |
| i=1 | A[1] |
| i=2 | A[2] |
| i=3 | A[3] |

Case 1:
Worst Case

| | |
|---|---|
| A[0]=**1** | ➡ M=1 |
| A[1]=**2** | ➡ M=2 |
| A[2]=**3** | ➡ M=3 |
| A[3]=**4** | ➡ M=4 |

4 updates of M

Case 2:
Best Case

| | |
|---|---|
| A[0]=**4** | ➡ M=4 |
| A[1]=**3** | |
| A[2]=**2** | |
| A[3]=**1** | |

1 update of M

```
for (i=0; i<n; ++i) {
    if (A[i] >= M) {      // always n comparisons
        M = A[i];         // the updating of M depends on the data
    }                     // minimum 1 update, maximum n updates
```

https://discrete.gr/complexity/

# Assignment instruction counts

```
M = A[0];                       // 2 instructions
for (i=0; i<n; ++i) {
    if (A[i] >= M) {
        M = A[i];               // 2 instructions
    }
}
```

A[0] – **1** instruction
M = – **1** instruction

A[i] – **1** instruction
M = – **1** instruction

https://discrete.gr/complexity/

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

}
```

```
for (i=0; i<n; ++i) {
    if (A[i] >= M) {
        M = A[i];
}
```

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

}
```

Initialization * **1**

Loop body * **n**

Update * **n**

https://discrete.gr/complexity/

# **for** loop instruction counts

```
for (i=0; i<n; ++i) {
    if (A[i] >= M) {
        M = A[i];
    }
```

Initialization * **1**

| | |
|---|---|
| i=0 | : **1** instruction |
| i<n | : **1** instruction |

Update * **n**

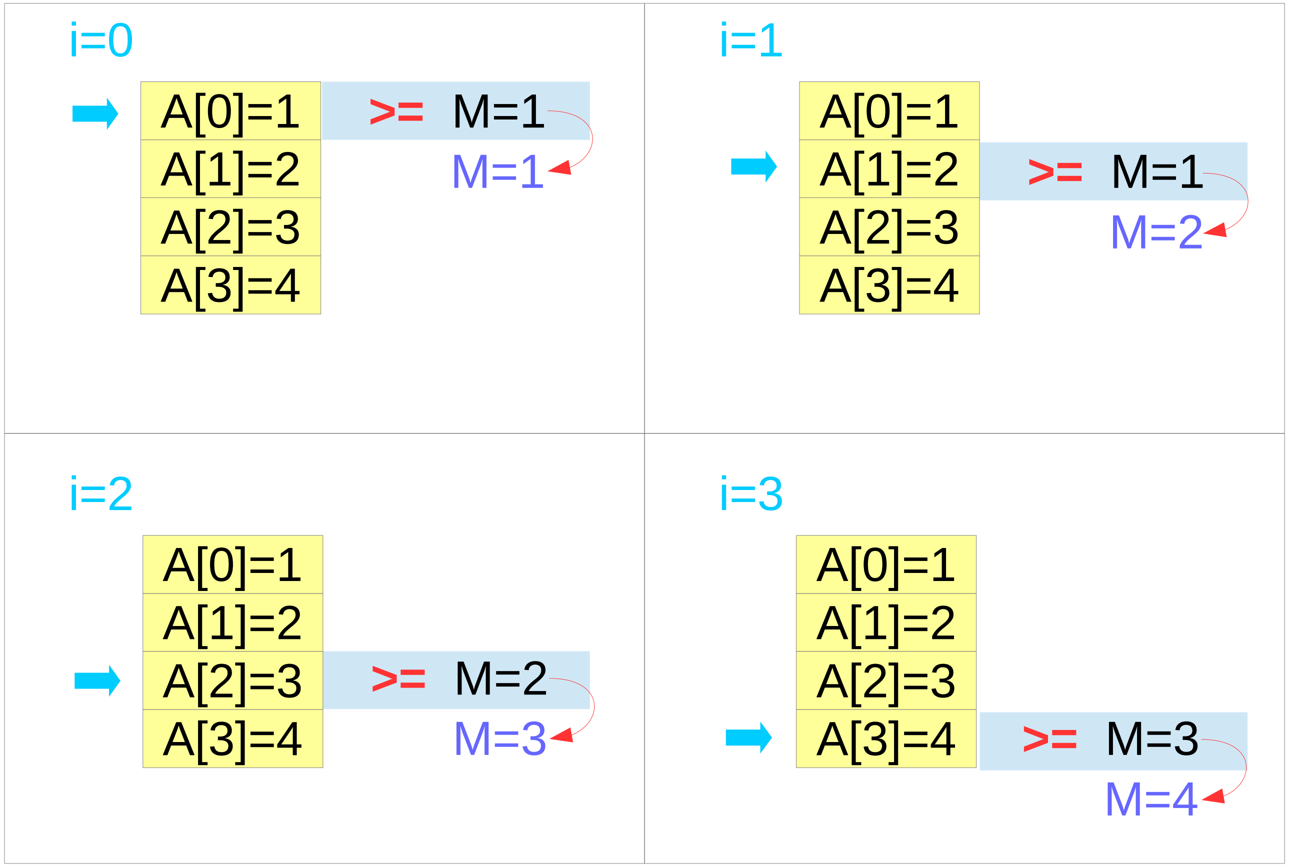
| | |
|---|---|
| ++i | : **1** instruction |
| i<n | : **1** instruction |

Loop body * **n**

| | | | |
|---|---|---|---|
| A[i] | : **1** instruction | *** n** | always |
| >= M | : **1** instruction | | |
| A[i] | : **1** instruction | *** (1~ n)** | depending on the input data |
| M= | : **1** instruction | | |

https://discrete.gr/complexity/

# Worst case examples




```
for (i=0; i<n; ++i) {
    if (A[i] >= M) {
        M = A[i];
    }
```

**2n + 2n = 4n**

instructions

**n** comparisons

**n** updates

https://discrete.gr/complexity/

# Best case examples


```
for (i=0; i<n; ++i) {
    if (A[i] >= M) {
        M = A[i];
    }
```

**2n + 2**

instructions

**n** comparisons

**1** update

https://discrete.gr/complexity/

# Asymptotic behavior

```
M = A[0]; ----------------- 2       instructions
for (i=0; i<n; ++i) { -------- 2 + 2n   instructions (init + update)
    if (A[i] >= M) { -------- 2n      instructions
        M = A[i]; --------- 2 ~ 2n   instructions
    }
}
```

$f(n) = \begin{cases} 6n+4 & \text{instructions for the worst case} \\ 4n+6 & \text{instruction for the best case} \end{cases}$

$f(n) = \Theta(n)$

$f(n) = O(n)$

$f(n) = \Omega(n)$

https://discrete.gr/complexity/

# Θ(n) codes

```
// Here c is a positive integer constant
for (i = 1; i <= n; i += c) {
    // some Θ(1) expressions
}

for (int i = n; i > 0; i -= c) {
    // some Θ(1) expressions
}
```

https://stackoverflow.com/questions/11032015/how-to-find-time-complexity-of-an-algorithm

# Θ(n) codes

```
for (i = 1; i <= n; i += c)
```

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $c=1$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | $\cdots$ | $\approx n$ | $=\Theta(n)$ |
| $c=2$ | 1 | | 3 | | 5 | | 7 | | 9 | | 11 | | 13 | | 15 | | $\cdots$ | $\approx n/2$ | $=\Theta(n)$ |
| $c=3$ | 1 | | | 4 | | | 7 | | | 10 | | | 13 | | | 16 | $\cdots$ | $\approx n/3$ | $=\Theta(n)$ |
| $c=4$ | 1 | | | | 5 | | | | 9 | | | | 13 | | | | $\cdots$ | $\approx n/4$ | $=\Theta(n)$ |

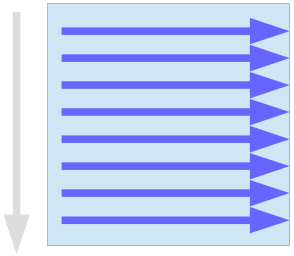
```
for (int i = n; i > 0; i -= c) {
```

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $c=1$ | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | $\cdots$ | $\approx n$ | $=\Theta(n)$ |
| $c=2$ | 16 | | 14 | | 12 | | 10 | | 8 | | 6 | | 4 | | 2 | | $\cdots$ | $\approx n/2$ | $=\Theta(n)$ |
| $c=3$ | 16 | | | 13 | | | 10 | | | 7 | | | 4 | | | 1 | $\cdots$ | $\approx n/3$ | $=\Theta(n)$ |
| $c=4$ | 16 | | | | 12 | | | | 8 | | | | 4 | | | | $\cdots$ | $\approx n/4$ | $=\Theta(n)$ |

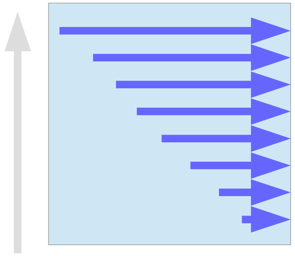
https://stackoverflow.com/questions/11032015/how-to-find-time-complexity-of-an-algorithm

# Θ(n²) codes

```
for (i = 1; i <=n; i += c) {
   for (j = 1; j <=n; j += c) {
      // some Θ(1) expressions
   }
}
```



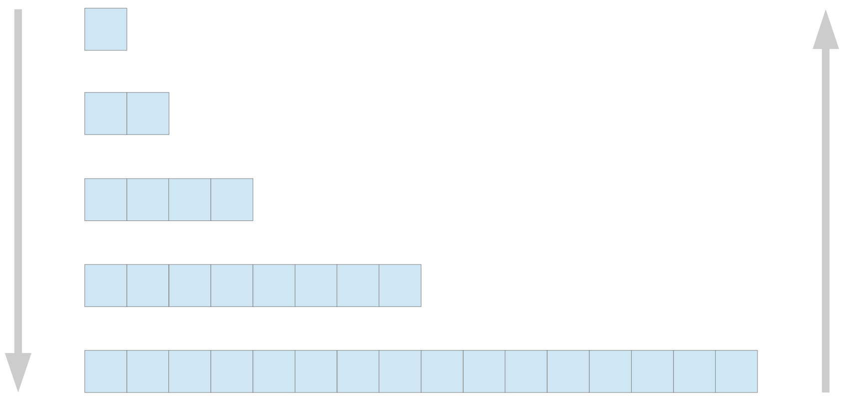
```
for (i = n; i > 0; i -= c) {
   for ( j = i+1; j <=n; j += c) {
      // some Θ(1) expressions
}
```



https://stackoverflow.com/questions/11032015/how-to-find-time-complexity-of-an-algorithm

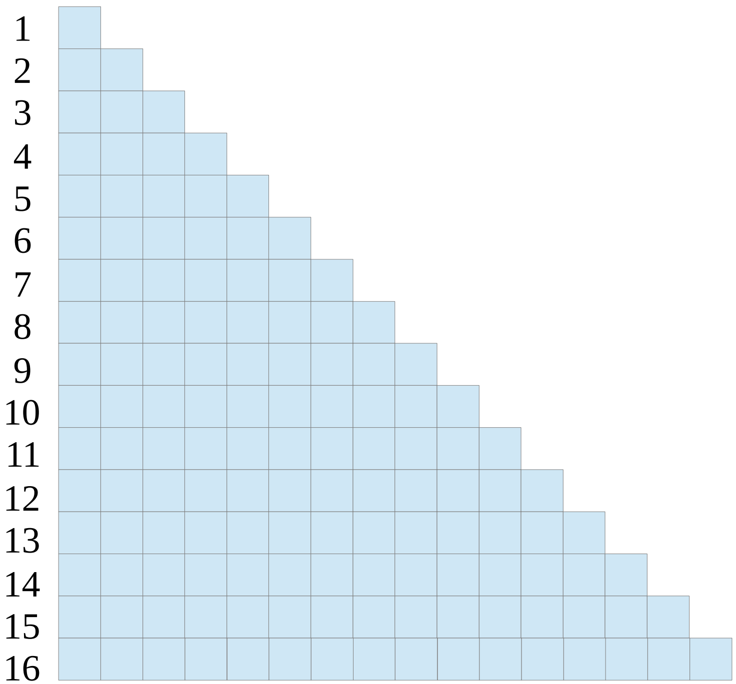
# Θ(log **n**) codes

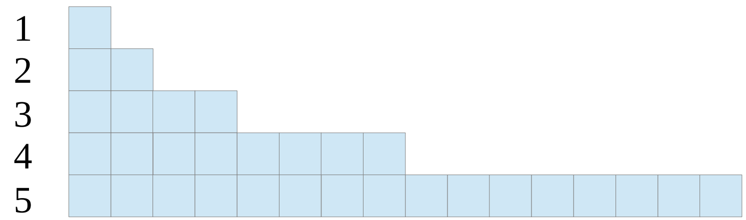
```
for (int i = 1; i <=n; i *= c) {
   // some Θ(1) expressions
}
for (int i = n; i > 0; i /= c) {
   // some Θ(1) expressions
}
```



https://stackoverflow.com/questions/11032015/how-to-find-time-complexity-of-an-algorithm

# Θ(**n**) vs. Θ(log **n**)




https://stackoverflow.com/questions/11032015/how-to-find-time-complexity-of-an-algorithm

# Θ(log **n**) codes

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {          // i = i^c
   // some Θ(1) expressions
}
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = fun(i)) {             // i = i^(1/c)
   // some Θ(1) expressions
}
```

$i = i^2, i = i^3$

https://stackoverflow.com/questions/11032015/how-to-find-time-complexity-of-an-algorithm

# Θ(log log **n**) codes

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {          // i = i^c
  // some Θ(1) expressions
}
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = fun(i)) {             // i = i^(1/c)
  // some Θ(1) expressions
}
```

$i = i^2 \quad (2, 2^2, 2^4, 2^8, 2^{16}, \cdots)$

$i = i^{\frac{1}{2}} \quad (n, n^{\frac{1}{2}}, n^{\frac{1}{4}}, n^{\frac{1}{8}}, n^{\frac{1}{16}}, \cdots)$

https://stackoverflow.com/questions/11032015/how-to-find-time-complexity-of-an-algorithm

# Θ(log log **n**) codes

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {        // i = i^c
   // some Θ(1) expressions
}
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = fun(i)) {            // i = i^(1/c)
   // some Θ(1) expressions
}
```

$i = i^2 \quad (2, 2^2, 2^4, 2^8, 2^{16}, \cdots)$

$i = i^{\frac{1}{2}} \quad (n, n^{\frac{1}{2}}, n^{\frac{1}{4}}, n^{\frac{1}{8}}, n^{\frac{1}{16}}, \cdots)$

https://stackoverflow.com/questions/11032015/how-to-find-time-complexity-of-an-algorithm

# Some Algorithm Complexities and Examples (1)

**Θ(1) – Constant Time**

not affected by the input size **n**.

**Θ(n) – Linear Time**

Proportional to the input size **n**.

**Θ(log n) – Logarithmic Time**

recursive subdivisions of a problem

binary search algorithm

**Θ(n log n) – Linearithmic Time**

Recursive subdivisions of a problem and then merge them

merge sort algorithm.

https://stackoverflow.com/questions/11032015/how-to-find-time-complexity-of-an-algorithm

# Some Algorithm Complexities and Examples (2)

**$\Theta(n^2)$ – Quadratic Time**

bubble sort algorithm

**$\Theta(n^3)$ – Cubic Time**

straight forward matrix multiplication

**$\Theta(2^n)$ – Exponential Time**

Tower of Hanoi

**$\Theta(n!)$ – Factorial Time**

Travel Salesman Problem (TSP)

https://stackoverflow.com/questions/11032015/how-to-find-time-complexity-of-an-algorithm

## References


[1] http://en.wikipedia.org/
[2]