SCC.121: ALGORITHMS AND COMPLEXITY
Big-O Notation

Emma Wilson e.d.wilson1@lancaster.ac.uk

Today’s Lecture

Aim: To introduce big-O notation

Learning objectives:

- To know how to calculate the complexity of example algorithms in big-O notation
- To be able to calculate the complexity of algorithms using big-O notation without operation counting (more next lecture)

Outline

- Formal definition of big-O
- Big-O notation in general

Outline

- **Formal definition of big-O**
- Big-O notation in general

Recap: Growth rate of functions

Listed from slowest to fastest growth:

- **1** → Constant growth
- **log n** → Logarithmic growth
- **n^c** → where $0<c<1$
- **n** → Linear growth
- **n log n**
- **n^2** → Quadratic growth
- **n^2 log n**
- **n^3** → Cubic growth
- **n^c** → Polynomial growth (c is a constant number)
- **2^n** → Exponential growth
- **3^n** → Exponential growth
- **c^n** → Exponential growth (c is a constant number)
- **n!** → Factorial growth

The Big-O Notation

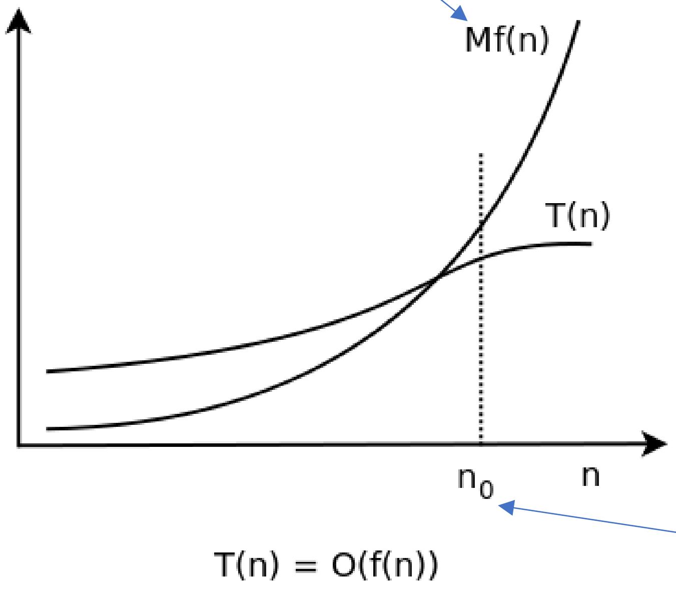
The growth of functions is usually described using the **big-O notation**

- **The formal mathematical definition of Big O:**
  - Let T(n) and f(n) be two positive functions from the integers or the real numbers to the real numbers
  - We write **$T(n) \in O(f(n))$**, and say that $T(n)$ has order of $f(n)$, if there are positive constants **M** and **n_0** such that
    - $T(n) \leq M \times f(n)$ for all $n \geq n_0$

The Big-O Notation

This graph shows a situation where all of the conditions in the definition are met

M is a positive constant



n_0 is a positive constant

- T(n) is O(f(n)) if even as n becomes arbitrarily large, T(n)'s growth is bounded from **<u>above</u>** by f(n), meaning it grows no faster than f(n)

The Big-O Notation

The idea behind the big-O notation is to establish an **upper boundary** for the growth of the function **T(n)** for large **n**

This boundary is specified by the function ***f(n)*** that is usually much **simpler** than **T(n)**

- For example, $f(n) = 1, f(n) = \log_2 n, f(n) = n, f(n) = n\log_2 n, f(n) = n^2, f(n) = n^3, \dots, f(n) = 2^n, f(n) = 3^n, \dots, f(n) = n!$

The Big-O Notation Example#1

Example#1:

- $T(n) = 3n + 4$
- $f(n) = n$
- Show that $T(n)$ is $O(f(n))$ which means $T(n)$ is $O(n)$

We need to find an M and n_0 such that

$$T(n) \leq M \times f(n) \text{ for all } n \geq n_0$$

The Big-O Notation Example#1

Example#1:

- $T(n) = 3n + 4$
- $f(n) = n$
- Show that $T(n)$ is $O(f(n))$ which means $T(n)$ is $O(n)$
- We need to find an M and n_0 such that: $T(n) \leq M \times n$ for all $n \geq n_0$

Solution:

- For $n \geq 1$ we have: $T(n) = 3n + 4 \leq 3n + 4n$
- So, $T(n) = 3n + 4 \leq 7n$
- Therefore, for $M=7$ and $n_0=1$ $\rightarrow$ $T(n) \leq 7n$ for all $n \geq 1$
- $T(n) \in O(n)$

The Big-O Notation Example#2

Example#2:

- $T(n) = n^2 + 2n + 1$
- $f(n) = n^2$
- Show that $T(n)$ is $O(f(n))$ which means $T(n)$ is $O(n^2)$

We need to find an M and n_0 such that

$$T(n) \leq M \times f(n) \text{ for all } n \geq n_0$$

The Big-O Notation Example#2

- **Example#2:**
  - $T(n) = n^2 + 2n + 1$
  - $f(n) = n^2$
  - Show that $T(n)$ is $O(f(n))$ which means $T(n)$ is $O(n^2)$
  - We need to find an M and n_0 such that: $T(n) \leq M \times n^2$ for all $n \geq n_0$

- **Solution:**
  - For $n \geq 1$ we have: $T(n) = n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2$
  - So, $T(n) = n^2 + 2n + 1 \leq 4n^2$
  - Therefore, for $M=4$ and $n_0=1$ $\rightarrow$ $T(n) \leq 4 \times n^2$ for all $n \geq 1$
  - $T(n) \in O(n^2)$

The Big-O Notation Example#3

Example#3:

- $T(n) = 3\log_2 n + 3$
- $f(n) = \log_2 n$
- Show that T(n) is O(f(n)) which means T(n) is $O(\log_2 n)$
- We need to find an M and n_0 such that: $T(n) \le M \times \log_2 n$ for all $n \ge n_0$

Solution:

- For $n \ge 2$ we have: $T(n) = 3\log_2 n + 3 \le 3\log_2 n + 3\log_2 n$
- So, $T(n) = 3\log_2 n + 3 \le 6\log_2 n$
- Therefore, for M=6 and $n_0=2$ ➔ $T(n) \le 6\log_2 n$ for all $n \ge 2$
- $T(n) \in O(\log_2 n)$

The Big-O Notation Example#3

We can follow as similar approach to also show that $\log_2 \mathrm{n}$ is O($\log_b \mathrm{n}$) : lab exercise

- So, in the **Example#3:**
- T(n) = $3\log_2 \mathrm{n} + 3$
- f(n) =$\log_2 \mathrm{n}$
- We can say T(n) is O($\log_b \mathrm{n}$) or <u>in general</u> **O($\boldsymbol{\log n}$)**

The Big-O Notation
Example#4

Example#4:

- T(n) = 30
- f(n) = 1
- Show that T(n) is O(f(n)) which means T(n) is O(1)
- We need to find an M and n_o such that: T(n) ≤ M for all n ≥ n_o

Solution:

- We have: T(n) = $30 \leq 31$
- So, T(n) = $30 \leq 31 \times 1$
- Therefore, for M=31 ➔ T(n) $\leq 31\, f(n)$ for all n
- T(n) ∈ $O(1)$

Outline

- Formal definition of big-O
- Big-O notation in general

Outline

- Formal definition of big-O
- **Big-O notation in general**

The big O notation in general

Constant $< \log n < n^c (0 < c < 1) < n < n \log n < n^2 < n^3 \ldots < 2^n < 3^n < \ldots < n!$

- T(N) = $C_1 \times N + C_0$ **O(N)**
- T(N) = $C_2 \times N^2 + C_1 \times N + C_0$ **O(N^2)**
- T(N) = $C_3 \times N^3 + C_2 \times N^2 + C_1 \times N + C_0$ **O(N^3)**
- T(N) = $C_k \times N^k + C_{k-1} \times N^{k-1} + \cdots + C_1 \times N + C_0$ **O(N^k)**

More examples:

- T(N) = $C_2 \times N + C_1 \log N + C_0$ → O(N) N is dominant term
- T(N) = $C_2 \times N^{1000} + C_1 2^N + C_0$ → $O(2^N)$ 2^N is dominant term

Big-O: Example Notation

- Constant : **O(1)**
- Logarithmic: **O(log n)**
- Linear: **O(n)**
- Quadratic **$O(n^2)$**
- Polynomial (c a constant number): **$O(n^c)$**
- Exponential (c a constant number): **$O(c^n)$**
- Factorial: **O(n!)**

The Growth of Functions Questions

- Which function grows faster?
  - $T_1(\mathrm{n}) = \ 1000n^2$
  - $T_2(\mathrm{n}) = n \log n + 5000n$

- Which function grows faster?
  - $T_1(\mathrm{n}) = \ 1000 \times 2^n$
  - $T_2(\mathrm{n}) = n!$

- Which function grows faster?
  - $T_1(\mathrm{n}) = \ n^{0.1}$
  - $T_2(\mathrm{n}) = \log n + 10$

slido

Growth of functions

ⓘ Start presenting to display the poll results on this slide.

The Growth of Functions

Constant $< \log n <\ n^c (\text{where } 0 < c < 1) < n < n \log n <$ $n^2 < n^3 \ldots <\ 2^n < 3^n <\ \ldots < n!$

Lancaster University

- Which function grows faster?
  - $T_1(\text{n}) =\ 1000n^2$ → $O(n^2)$
  - $T_2(\text{n}) = n \log n + 5000n$ → $O(n \log n)$

- Which function grows faster?
  - $T_1(\text{n}) =\ 1000 \times 2^n$ → $O(2^n)$
  - $T_2(\text{n}) = n!$ → $O(n!)$

- Which function grows faster?
  - $T_1(\text{n}) =\ n^{0.1}$ → $O(n^{0.1})$
  - $T_2(\text{n}) = \log n + 10$ → $O(\log n)$

The Big-O notation
General Question

- If $T(n)$ is $O(n^2)$, is it also $O(n^3)$?

slido

If T(n) is O(n^2), is it also O(n^3)?

ⓘ Start presenting to display the poll results on this slide.

The Big-O Notation Note

Lancaster University

Question: If $T(n)$ is $O(n^2)$, is it also $O(n^3)$?

Yes. Why?

- $T(n)$ is $O(n^2)$ means $T(n) \le M \times n^2$ for all $n \ge n_o$
- We now need to prove $T(n)$ is $O(n^3)$ which means $T(n) \le M \times n^3$ for all $n \ge n_o$

- Since n^3 grows faster than n^2 → $n^2 \le n^3$ for $n \ge 1$
- Thus $T(n) \le M \times n^2 \le M \times n^3$ → $T(n)$ is $O(n^3)$

In practice, we always use **smallest** simple function $f(n)$ for which $T(n)$ is $O(f(n))$

The Growth of Functions (Questions)

- Which function grows faster?
  - $T_1(\text{n}) = 1000n^2 \rightarrow O(n^2), O(n^3), O(n^4), \ldots, O(2^n), \ldots, O(n!)$
  - $T_2(\text{n}) = n \log n + 5000n \rightarrow O(n \log n), O(n^2), \ldots, O(2^n), \ldots, O(n!)$

- Which function grows faster?
  - $T_1(\text{n}) = 10002^n \rightarrow O(2^n), O(3^n), \ldots, O(n!)$
  - $T_2(\text{n}) = n! \rightarrow O(n!), O(n^n)$

- Which function grows faster?
  - $T_1(\text{n}) = n^{0.1} \rightarrow O(n^{0.1}), O(n^2), O(n^3), O(n^4), \ldots, O(2^n), \ldots, O(n!)$
  - $T_2(\text{n}) = \log n + 10 \rightarrow O(\log n), O(n), \ldots, O(2^n), \ldots, O(n!)$

The smallest simple functions are shown in RED

Dominant Terms: More examples

Given the processing time T(n) spent by an algorithm for solving a problem of size n

- Find the dominant term(s) and specify the **lowest** Big-O complexity

Expression	**Dominant term(s)**	**O(...)**
$5 + 0.001n^3 + 0.025n$	$0.001n^3$	O(n^3)
$500n + 100n^{1.5} + 50n\log_{10} n$	$100n^{1.5}$	O($n^{1.5}$)
$0.3n + 5n^{1.5} + 2.5n^{1.75}$	$2.5n^{1.75}$	O($n^{1.75}$)
$n^2 \log_2 n + n(\log_2 n)^2$	$n^2 \log_2 n$	O($n^2 \log n$)
$0.01n\log_2 n + n(\log_2 n)^2$	$n(\log_2 n)^2$	O($n(log\, n)^2$)
$\log_2 n + \log_2 \log_2 n$	$\log_2 n$	O($\log n$)

$$\text{constant} < \log n < n^c (\text{where } 0 < c < 1) < n < n\log n < n^2 < n^3 \ldots < 2^n < 3^n < \ldots < n!$$

Single loops with O(1) instructions

Loop running constant times: **O(1)**

- Loop runs constant times, performing O(1) operations at each iteration
- Time complexity = c*O(1) = O(1)

```
// c is a constant
for (int i = 0; i <= c; i++) {
        //O(1) instructions
}
```

Loop incrementing/ decrementing by constant c: **O(n)**

- Loop runs n/c times, performing O(1) operations at each iteration
- Time complexity = 1/c *O(n)* O(1) = O(n)

```
// c is a constant
for (int i = 0; i <= n; i+=c)
{
        //O(1) instructions
}
```

Loop divided/ multiplied by constant c: **O(log n)**

- Loop runs $\log_c(n)$ times, performing O(1) operations at each iteration
- Time complexity = $\log_c(n)$ * O(1) = O(log n)

```
// c is a constant
for (int i = 1; i <= n; i*=c)
{
        //O(1) instructions
}
```

Single loops with O(f(n)) instructions

Loop running constant times:

- Loop runs constant times, performing O(1) operations at each iteration
- Time complexity = c*O(f(n)) = **O(f(n))**

```
// c is a constant
for (int i = 0; i <= c; i++) {
        //O(f(n)) instructions
}
```

Loop incrementing/ decrementing by some constant c:

- Loop runs n/c times, performing O(f(n)) operations at each iteration
- Time complexity = 1/c *O(n)* O(f(n)) = **O(n*f(n))**

```
// c is a constant
for (int i = 0; i <= n; i+=c)
{
        //O(f(n)) instructions
}
```

Loop divided/ multiplied by some constant c:

- Loop runs $\log_c(n)$ times, performing O(f(n)) operations at each iteration
- Time complexity = $\log_c(n)$ * O(f(n)) = O(log n*f(n))

```
// c is a constant
for (int i = 1; i <= n; i*=c)
{
        //O(f(n)) instructions
}
```

Nested Loops

- Complexity of nested loops equal to the number of times innermost statement executed*complexity of statement
- Complexity of inner loop*complexity of outer loop
- Care needed if loops are not independent

```
for (int i = 0; i < n; i = i+1 )\\
{
        for (int j = 0; j < n; j = j + 1)
        {
                \\some O(f(n)) expressions
        }
}
```

Example: Inner loop runs n times for every iteration of outer loop

- Total number of nested loop iterations: $O(n)*O(n) = O(n^2)$
- At each iteration nested loop doing an $O(f(n))$ operation
- Overall time complexity = $O(f(n))*O(n^2)$ = $\mathbf{O(n^2 * f(n))}$

Care with general rules – check code!

```
// c is a constant
for (int i = 0; i <= n; i*=c)
{
        //O(f(n)) instructions
}
```

```
// c is a constant
for (int i = n;i>-1; i/=2) {
        //O(f(n)) instructions
}
```

- Rules are simple, but **care** needed!

Summary

Today's lecture: looked at using big O notation

- The growth of functions is usually described using the big-O notation
- Can calculate big-O from the term that grows the fastest (dominant term) in T(n)
- In practice, we always use the ***smallest*** simple function f(n) for which T(n) is O(f(n))

- **Next:** big-O examples and big-omega and big-theta.