Asymptotic Notations and Basic Efficiency Classes

Asymptotic Notation

- **1. Purpose**

- Asymptotic notation provides a mathematical way to describe the efficiency of an algorithm — especially how its running time or space requirements grow as the input size (n) increases.
- **It helps us:**
- Compare algorithms independently of hardware and programming language.
- Focus on the rate of growth rather than exact time.
- Identify upper and lower bounds of performance (best, average, worst cases).

2. Why Asymptotic Analysis?

Instead of measuring actual time (which depends on processor, compiler, and system speed), we analyze how the **running time scales with input size.**

So, we use a mathematical function f (n) that expresses the number of **basic operations** performed by the algorithm as a function of input size n.

3. Cases of Analysis

For any algorithm, we can describe performance in three typical cases:

Case	Meaning	Example (Linear Search)	Function
Best Case	Minimum number of operations	Target is first element	$\Omega(n)$
Average Case	Expected number of operations	Target somewhere in middle	$\Theta(n)$
Worst Case	Maximum number of operations	Target not in array	$O(n)$

4. Main Asymptotic Notations

Notation	Definition	Describes
Big O (O)	Upper bound on growth rate	Worst-case complexity
Big Omega (Ω)	Lower bound on growth rate	Best-case complexity
Big Theta (Θ)	Tight bound on growth rate	Average-case / Exact order

- **What are different asymptotic notations?**

1. Definition

Asymptotic Notation is a mathematical tool used to describe the **running time** or **space requirement** of an algorithm **in terms of input size** n— especially as $n$becomes very large (approaches infinity).

It allows us to:

- Compare algorithms **independent of hardware or coding language**.
- Focus on **growth rate** rather than exact time.
- Express **upper, lower,** and **tight** bounds on performance.

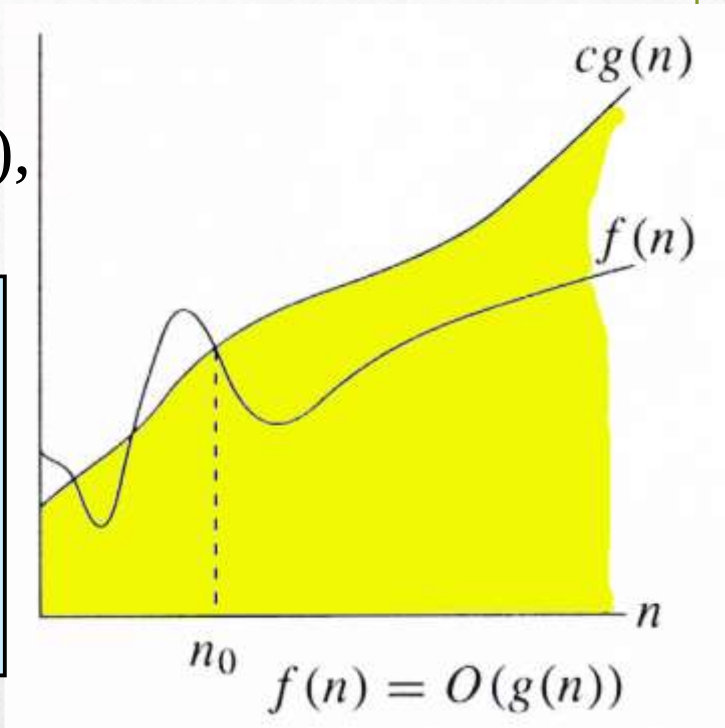
Asymptotic Notation

- Θ , O ,Ω
- Defined for functions over the natural numbers.
  - **Ex:** $f)n = (\Theta)n.(^2$
    - Describes how $f)n$ ot nosirapmoc ni sworg $(n.^2$
- Define a ***set*** ot desu ecitcarp ni ;snoitcnuf fo .sezis noitcnuf owt erapmoc
- The notations describe different rate-of-growth relations between the defining function and the defined set of functions.

O-notation

For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$O(g(n)) = \{f(n) :$
$\exists$ positive constants c and n_0,
such that $\forall n \geq n_0$,
we have $0 \leq f(n) \leq cg(n)$ $\}$



Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

$g(n)$ is an *asymptotic upper bound* for $f(n)$.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$.
$\Theta(g(n)) \subset O(g(n))$.

Example 1 : f(n)=4n3+10*4n2+5n+1
(usually written as $f(n) = 4n^3 + 40n^2 + 5n + 1$)
We want to find a function $g(n)$such that

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

Step 1: Choose the dominant term
Among $n^3, n^2, n, 1$,
the **highest power of n** dominates as $n$becomes large.
So, we choose

$$g(n) = n^3$$

Step 2: Find constants c and n_0

We want constants $c > 0$ and $n_0 > 0$ such that:

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

Substitute:

$$4n^3 + 40n^2 + 5n + 1 \leq c \cdot n^3$$

Divide both sides by n^3:

$$4 + \frac{40}{n} + \frac{5}{n^2} + \frac{1}{n^3} \leq c$$

As n increases, the fractions become very small.

For $n \geq 2$, this inequality is true if we choose $c = 5$.

Step 3: Conclusion

For $n \geq 2$,

$$f(n) \leq 5n^3$$

Hence,

$$f(n) = O(n^3)$$

✓ **Final Answer:**

The time complexity of $f(n)$ is **O(n³)**.

Example 2 :We are given:

$$f(n) = 3n + 8$$

We claim that $f(n) = O(n)$.

That means we need to find constants $c > 0$and $n_0 > 0$such that:

$$f(n) \leq c \cdot n \; for\, all\, n \geq n_0$$

Step 1: Substitute

We test if

$$3n + 8 \leq c \cdot n$$

We choose $c = 4$.

Then:

$$3n + 8 \leq 4n$$

Simplify:

$$8 \leq n$$

Step 2: Find n_0

This inequality holds **whenever** $n \geq 8$.

Hence, we can take:

$$c = 4, n_0 = 8$$

✓ **Conclusion:**

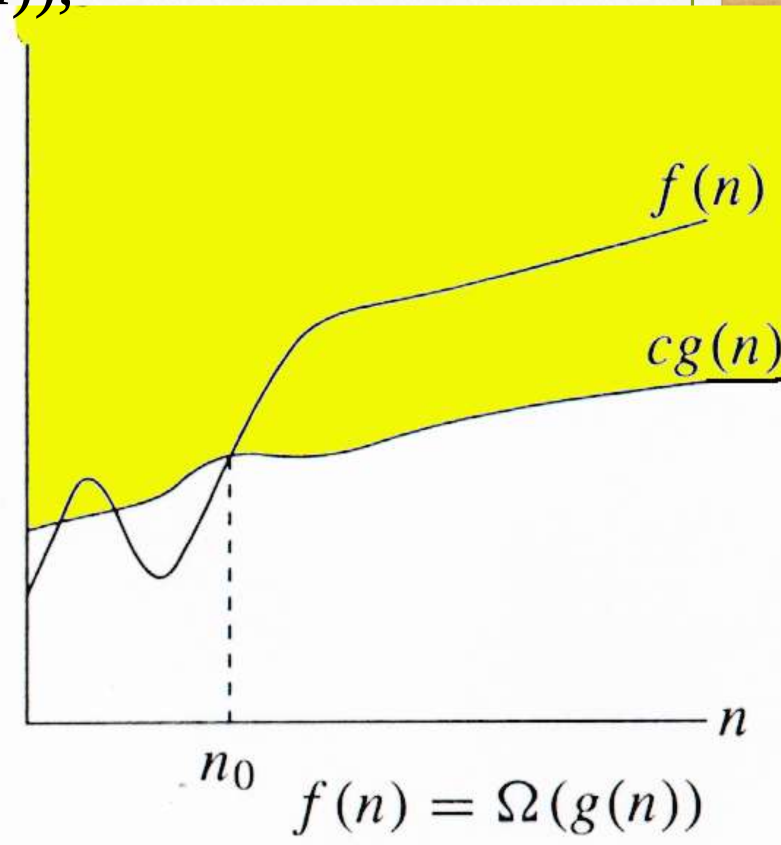
$$f(n) = 3n + 8 = O(n)$$

with constants $c = 4, n_0 = 8$.

Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$\Omega(g(n)) = \{f(n) :$
$\exists$ **positive constants c and n_0,**
such that $\forall n \geq n_0$,
we have $0 \leq cg(n) \leq f(n)\}$



Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

$g(n)$ is an *asymptotic lower bound* for $f(n)$.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$.

$\Theta(g(n)) \subset \Omega(g(n))$.

Example 1: Find lower bound for $f(n) = 5n^2$

Step 1: Understand the Big-Omega definition

Big-Omega notation (Ω) describes an asymptotic lower bound. Formally, $f(n) = \Omega\big(g(n)\big)$ if there exist positive constants c and n_0 such that:

$$0 \leq c \cdot g(n) \leq f(n) \; for\, all\, n \geq n_0$$

Step 2: Choose an appropriate $g(n)$

For $f(n) = 5n^2$,we suspect it grows at least as fast as n. So let's try to prove $f(n) = \Omega(n)$.

Step 3: Set up the inequality

We need to find c and n_0 such that:

$$0 \leq c \cdot n \leq 5n^2 \; for\, all\, n \geq n_0$$

Step 4: Solve for the constants

$$c \cdot n \leq 5n^2$$

$$c \leq 5n$$

This inequality holds for all $n \geq 1$ if we choose $c = 1$, since $1 \leq 5n$ for all $n \geq 1$.

Step 5: Verify the solution

With $c = 1$ and $n_0 = 1$:

- For $n = 1$: $1 \cdot 1 = 1 \leq 5(1)^2 = 5$ ✓
- For $n > 1$: $1 \cdot n = n \leq 5n^2$ ✓

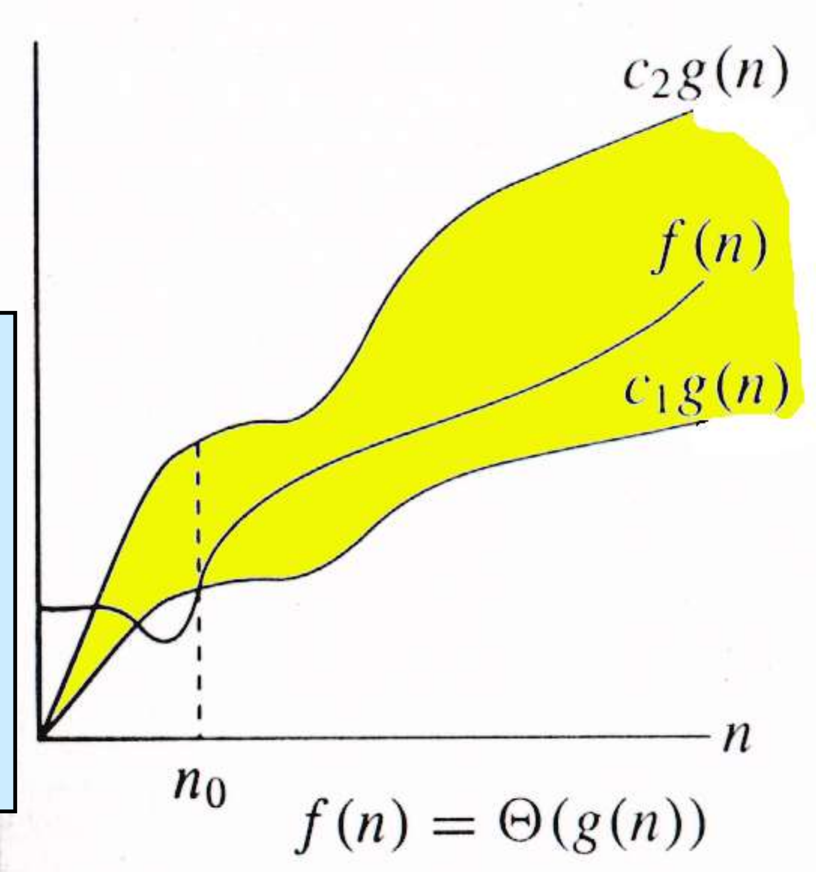
Conclusion:

$$5n^2 = \Omega(n) \quad \text{with } c = 1 \text{ and } n_0 = 1$$

Θ-notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n)$:
$\exists$ positive constants c_1, c_2, and n_0,
such that $\forall n \geq n_0$,
we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
$\}$



Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm

$g(n)$ is an *asymptotically tight bound* for $f(n)$.

Big-Theta (Θ) notation gives a **tight bound** on the growth rate of an algorithm.

If we say: $f(n)=\Theta(g(n))$

It means **f(n) grows at the same rate as g(n)** — not faster, not slower.

Think of **Θ(g(n))** as a **tight wrap** around the function **f(n)**.

- **From above:** $f(n) \leq c_2 \cdot g(n)$
- **From below:** $f(n) \geq c_1 \cdot g(n)$

So, for large values of n, f(n) stays **between** two constant multiples of g(n).

c1g(n)≤f(n)≤c2g(n)for all n≥n0

That's why we call it **"asymptotically tight bound."**

Example 1: Find Θ bound for $f(n) = \frac{n^2}{2} - \frac{n}{2}$

Step 1: Understand the Big-Theta definition

Big-Theta notation (Θ) describes an asymptotically tight bound. Formally, $f(n) = \Theta\big(g(n)\big)$ if there exist positive constants c_1, c_2, and n_0 such that:

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \ for\, all\, n \geq n_0$$

Step 2: Identify the candidate $g(n)$

For $f(n) = \frac{n^2}{2} - \frac{n}{2}$,the dominant term is n^2 ,so we suspect $f(n) = \Theta(n^2)$.

Step 3: Find the upper bound (c_2(

We need to show $f(n) \le c_2 \cdot n^2$:

$$\frac{n^2}{2} - \frac{n}{2} \le c_2 \cdot n^2$$

For $n \ge 1$, we can choose $c_2 = 1$:

$$\frac{n^2}{2} - \frac{n}{2} \le n^2 \checkmark$$

Step 4: Find the lower bound (c_1)

We need to show $c_1 \cdot n^2 \leq f(n)$:

$$c_1 \cdot n^2 \leq \frac{n^2}{2} - \frac{n}{2}$$

Let's try $c_1 = \frac{1}{5}$:

$$\frac{n^2}{5} \leq \frac{n^2}{2} - \frac{n}{2}$$

Multiply both sides by 10:

$$2n^2 \leq 5n^2 - 5n$$

$$0 \leq 3n^2 - 5n$$

$$n(3n - 5) \geq 0$$

This holds for all $n \geq 2$ (since $3n - 5 \geq 0$ when $n \geq \frac{5}{3}$).

For $n \geq 2$, the inequality holds.

Step 5: Verify the solution

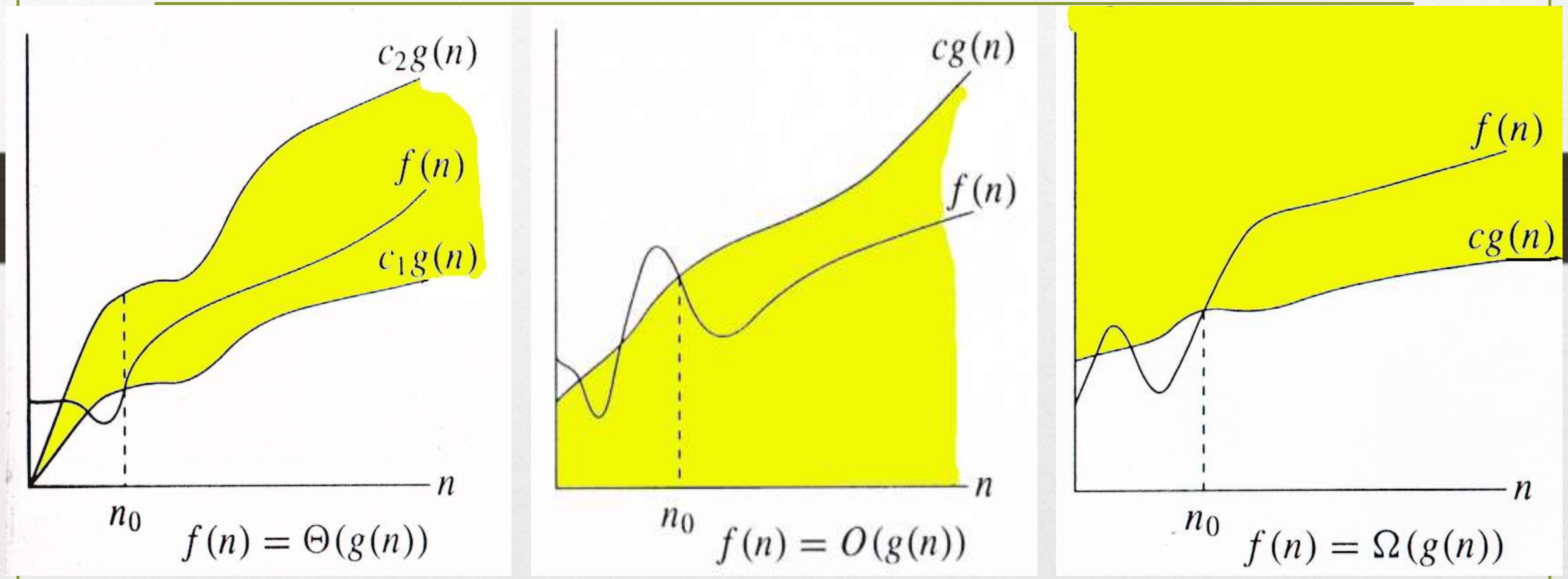
With $c = 1$ and $n_0 = 1$:

- For $n = 1$: $1 \cdot 1 = 1 \leq 5(1)^2 = 5$ ✓
- For $n > 1$: $1 \cdot n = n \leq 5n^2$ ✓

Conclusion:

$$5n^2 = \Omega(n) \quad \text{with } c = 1 \text{ and } n_0 = 1$$

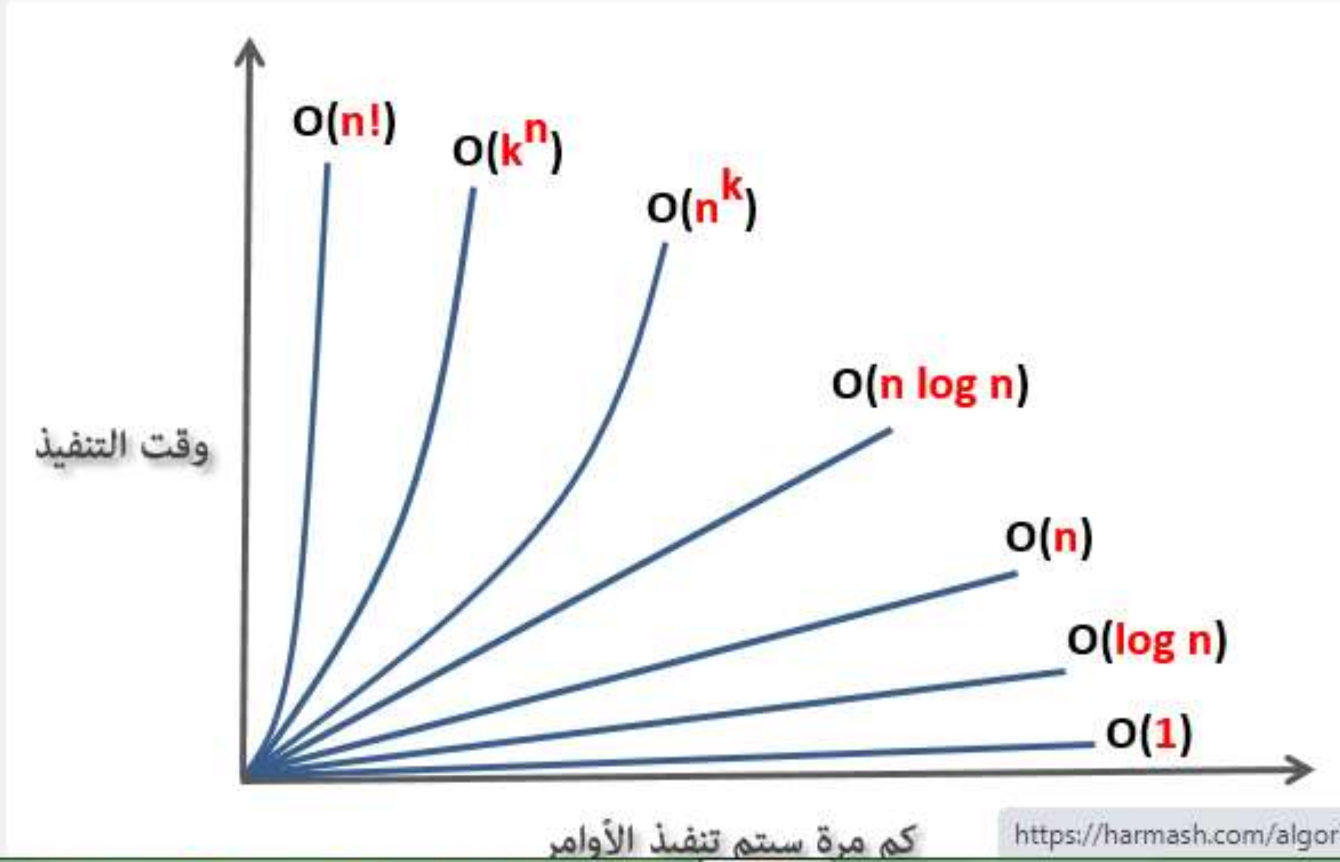
Relations Between Θ, *O*, Ω



تقييم أداء الخوارزميات

في هذا الدرس ستتعلم كيف تقوم بتقييم أداء الخوارزميات و كيف يتم قراءة التقييم لمعرفة ما إن كان فعال أم لا.
كما أننا سنعلمك كيف تستطيع حساب الوقت الذي يستغرقه الكود حتى يتنفذ بنفسك.

عندما نقوم بتقدير وقت تنفيذ أي خوارزمية (أكثر وقت تحتاجه) فإن النتيجة النهائية ستكون أحد النتائج المذكورة في الجدول التالي.



ما يجب أن تفهمه من الرسم أنه كلما كان الوقت الذي تستغرقه الخوارزمية منخفض كلما كان أداؤها أفضل.
و بالتالي المعادلة `O(1)` تعتبر الأفضل بينهم و المعادلة `O(n!)` تعتبر الأسوء على الإطلاق.

عند تقدير وقت الخوارزمية، يجب مقارنة أداءها على أساس الرسم السابق.
فمثلاً إذا وجدنا أداء الكود هو `O(n!)` فسنحاول إيجاد حل آخر يتنفذ بوقت أقل.

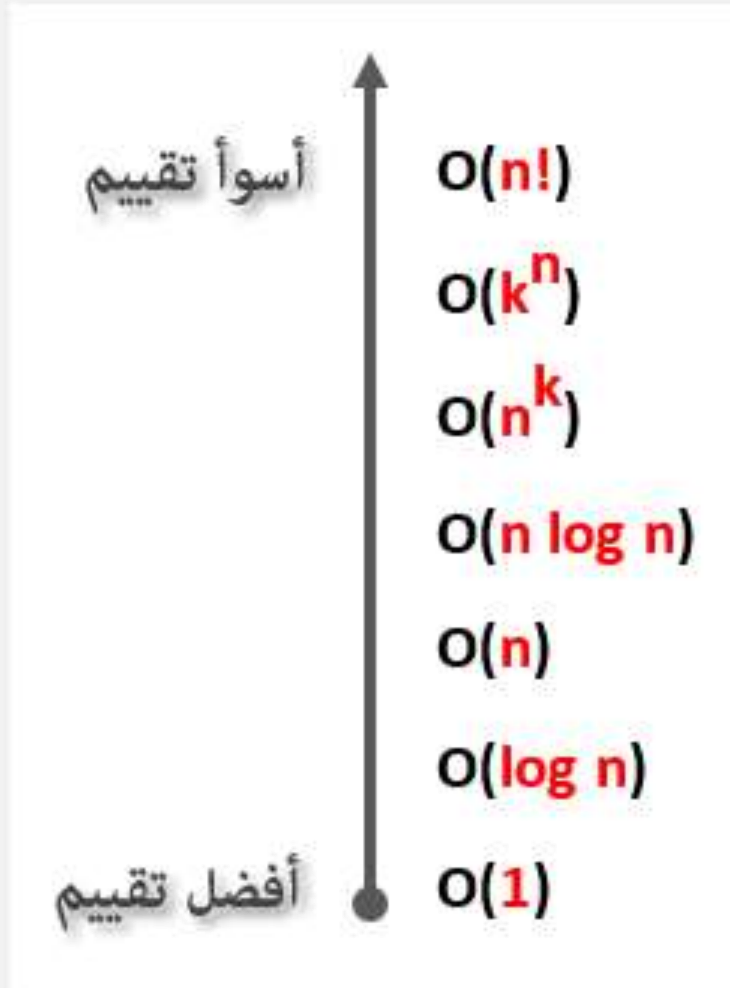
المعادلات التي تشير لتقييم جيد:

- `O(1)`
- `O(log n)`
- `O(n)`
- `O(n log n)`

المعادلات التي تشير لتقييم سيء:

- `O(nk)`
- `O(kn)`
- `O(n!)`

في الصورة التالية وضعنا ترتيب جميع المعادلات من الأفضل إلى الأسوأ بشكل أسهل لك في الحفظ.



مفهوم المعادلة O(1)

من المهم جداً معرفة أن الرقم 1 في هذه المعالة لا يشير للقيمة واحد, بل يعني أن كل أمر موضوع في الكود سيتنفذ مرة واحدة فقط.

بمعنى آخر, هذه المعادلة تعني أن الكود لا يحتوي على حلقات (**Loops**) و لا على دوال تستدعي نفسها (**Loops**).

إذا كان تقييم الكود هو O(1) فهذا يعني أنه ممتاز و يتنفذ بسرعة عالية جداً و لا يحتاج لأي تحسين.

و هو يعني أيضاً أن الوقت المتوقع لتنفيذ الكود ثابت (**Constant Time**) لا يتغير.

مثال

Python | Java | C# | C++ | C

```
def func():      # تعريف الدالة لا يحسب كخطوة

    a = 10       # إسناد القيمة يحسب خطوة
    b = 20       # إسناد القيمة يحسب خطوة
    s = a + b    # إسناد القيمة يحسب خطوة

    return s     # إرجاع القيمة يحسب خطوة

    # مجموع الأوامر التي ستنفذ هو 4 و لكن بما أنها مجرد أوامر عادية لا تتكرر أكثر من مرة فإنها لا تحسب إطلاقاً في معادلة الـ Big-O
```



```
Execution Steps = 1 + 1 + 1 + 1 = 4
Big O of any constant ==> O(1)
```

بما أن جميع الأوامر الموضوعة تتنفذ مرة واحدة فتقييم هذا الكود هو O(1) .

كما تلاحظ فإننا لا نهتم بعدد الخطوات المعروفة عند وضع التقييم النهائي بل نهتم بكم مرة ستكرر هذه الخطوات و في حال كانت لا تتكرر فإنها لا تدخل في التقييم.

مفهوم المعادلة O(n)

المتغير n في هذه المعالة يعني أن الكود سيتنفذ بعدد قيمة n كما هي الحال عندما نضع الكود بداخل حلقة.

بمعنى آخر, هذه المعادلة تعني أنه كلما كانت قيمة n أكبر, كلما كان الوقت الذي سيستغرقه تنفيذ الكود أكبر.

بما أن الوقت الذي تستغرقه هذه المعادلة يكبر بشكل متوازن مع كبر حجم الأوامر فهنا رسم التقييم سيكون خط مائل متوازن بينهما يسمى (**Linear Time**).

إذا كان تقييم الكود هو O(n) فهذا يعني أنه جيد و مقبول.

ملاحظة: إذا كان تقييم الكود جيد و لكن يمكن كتابته بطريقة أخرى أكثر بساطة و لا تتطلب استخدام حلقة, فالأولى أن نقوم بالتخلي عن الحلقة و اعتماد تلك الطريقة.

Python | Java | C# | C++ | C

```python
# تعريف الدالة لا يحسب كخطوة
def func(n):

    # إسناد أي قيمة يحسب خطوة واحدة و لكن عدد الخطوات غير مهم في تقييم الـ Big-O كما قلنا سابقاً
    s = 0

    # بما أنه عندنا حلقة تنفذ الكود الموضوع فيها (n times) - أي على حسب القيمة التي نضعها في n - سنضع المتغير n ضمن نتيجة الـ Big-O
    for i in range(1, n + 1):
        s += i

    # إرجاع القيمة يحسب خطوة واحدة و لكن عدد الخطوات غير مهم في تقييم الـ Big-O كما قلنا سابقاً
    return s
```

```
Execution Steps = 1 + n + 1
Execution Steps = 2 + n


Big O of 2 + n ==> O(n)
```

كما سبق و قلنا, الخطوات العادية أو الأوامر التي تتنفذ مرة واحدة لا تعتبر مهمة في تقييم أداء الكود.

عند تقييم الكود هنا لاحظ أننا لم نهتم إطلاقاً بعدد الخطوات الثابتة التي ستتنفذ, أي لم نهتم بالرقم `2` الذي يظهر في الـ **Step Execution** و لكننا إهتممنا فقط بالمتغير `n` الموضوع فيها.

عند تقييم أداء الكود فإننا دائماً ننظر لأعلى قيمة مجهولة ممكنة و في حالتنا هنا يعتبر المتغير `n` هو الأعلى لذلك كان التقييم النهائي لهذا الكود هو `O(n)` .

إذا كانت قيمة `n` مقسومة على رقم مثل `n / 2` هل ستتغير المعادلة؟

كلا لن تتغير, لأن قيمة `n` لا تزال مجهولة سواء كانت مقسومة أم لا و هذا ما سنراه في المثال التالي.

مفهوم المعادلة `O(log n)`

المقصود بالمعادلة `log n` هو عندما تتغير قيمة عداد الحلقة بشكل مضاعف أو مقسوم.

المثال الأول

في المثال التالي قمنا بجعل عدّاد الحلقة يتم ضربه بإثنين في كل دورة.

```python
# تعريف الدالة لا يحسب كخطوة
def func(n):

    # إسناد أي قيمة يحسب خطوة واحدة و لكن عدد الخطوات غير مهم في تقييم الـ Big-O كما قلنا سابقاً
    s = 0

    # بما أنه عندنا حلقة تنفذ الكود الموضوع فيها على حسب قيمة n و بنفس الوقت العداد يتم مضاعفة قيمته في كل دورة, سنضع log n ضمن نتيجة الـ Big-O
    i = 1
    while i <= n:
        s += i
        i *= 2

    # إرجاع القيمة يحسب خطوة واحدة و لكن عدد الخطوات غير مهم في تقييم الـ Big-O كما قلنا سابقاً
    return s
```

```
Execution Steps = 1 + log n + 1
Execution Steps = 2 + log n

Big O of 2 + log n ==> O(log n)
```

المثال الثاني

في المثال التالي قمنا بجعل عداد الحلقة يتم قسمته على اثنين في كل دورة.

مفهوم المعادلة $O(n^k)$

المقصود بالحرف `n` هو أن الكود موضوع بداخل حلقة.

المقصود بالحرف `k` أن الحلقة تحتوي أيضاً على حلقة أو أكثر بشكل متداخل (**Nested Loop**).

عند وضع التقييم للكود, الحرف `n` نضعه كما هو ليشير أنه يوجد حلقة, أما الحرف `k` فنضع مكانه عدد الحلقات المتداخلة و إليك بعض الأمثلة

- إذا كان الكود يتضمن حلقتين متداخلتين نكتب $O(n^2)$
- إذا كان الكود يتضمن ثلاث حلقات متداخلة نكتب $O(n^3)$
- إذا كان الكود يتضمن أربع حلقات متداخلة نكتب $O(n^4)$ و هكذا.

مفهوم المعادلة $O(n \log n)$

المقصود بهذا المعادلة أنه يوجد حلقتين متداخلتين. الحلقة الخارجية تتنفذ بمقدار `n` تماماً كالمعادلة $O(n)$. و الحلقة الداخلية تتنفذ نسبة لقيمة `n` أيضاً و لكن العداد
الخاص بها تتغير قيمته بشكل مضاعف أو مقسوم تماماً كالمعادلة $O(\log n)$.

مفهوم المعادلة $O(n!)$

المقصود بهذه المعادلة أن الدالة ستعيد إستدعاء نفسها بمقدار قيمة `n` .

ملاحظة: الكود الموضوع في المثال التالي تم شرحه بتفصيل ممل في دورة الخوارزميات و بالتحديد في درس تعريف دوال تستدعي نفسها

مفهوم المعادلة $O(k^n)$

المقصود بهذه المعادلة أن الدالة ستعيد إستدعاء نفسها بمقدار قيمة `n` و في كل عملية إستدعاء سيتم استدعاءها بشكل مضاعف أيضاً.

فعلى سبيل المثال، أول مرة تستدعي فيها نفسها، تقوم باستدعاء نفسها مرتين بشكل متوازي.

ثاني مرة تستدعي فيها نفسها، تقوم باستدعاء نفسها **4** مرات بشكل متوازي.

ثالث مرة تستدعي فيها نفسها، تقوم باستدعاء نفسها **8** مرات بشكل متوازي.

رابع مرة تستدعي فيها نفسها، تقوم باستدعاء نفسها **16** مرة بشكل متوازي و هكذا.

2. التعريف الرياضي لل Big O notaion:

لنفترض أننا حددنا كفاءة خوارزمية ما ب O(g(n)) ماذا يعني ذلك رياضيًا:

– يعني أن دالة وقت التنفيذ للخوارزمية تنتمي لمجموعة من الدوال .. بحيث أنه لكل دالة من هذه الدوال ولنسمي أي دالة مثلاً f(n) يوجد ثابت وليكن c بحيث أن g(n) *c > f(n) وذلك عندما n >= n0

سنحاول توضيح هذا المفهوم بمثال:

لنفترض أن هناك خوارزمية ما تتعامل مع مجموعة من البيانات عددها n ولحساب ال big o تبعًا للمفهوم الرياضي نقوم بعمل الخطوات التالية:

1. حساب وقت تنفيذ الخوارزمية وتمثيلها كدالة في المتغير n

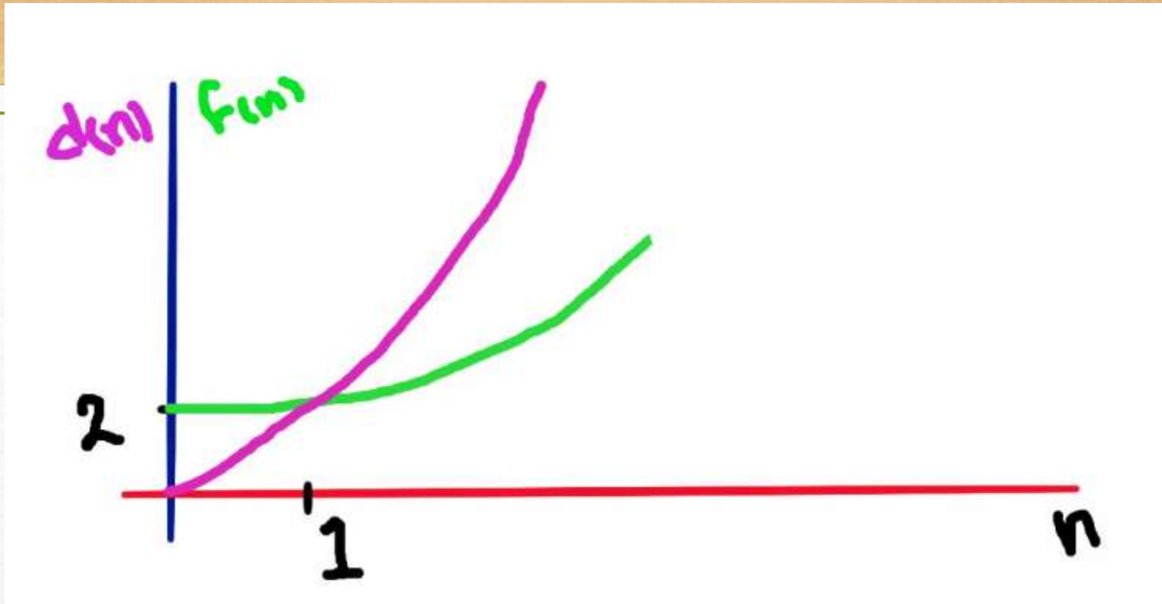
$$f(n) = 3 (n^2) + 2 n + 2$$

2. تحديد العامل المؤثر بشكل كبير في زيادة قيمة الدالة .. في الدالة السابقة واضح أنه عند القيم الكبيرة ل n يكون n^2 هو العامل المؤثر في الدالة

3. نحذف المتغيرات من المعادلة السابقة (في هذه الحالة المتغير n) ونقوم بضرب العامل المؤثر في كل الثوابت.. فإذا طبقنا ذلك على المعادلة السابقة ينتج لنا المعادلة التالية

$$d(n) = 3 (n^2) + 2 (n^2) + n^2 = 6 (n^2)$$

4. واضح من المعادلتين أن d(n) > f(n) عندما يكون المتغير n >= 1



وفي هذا الشكل يمكننا أن ترى أن الدالة d(n) تبدأ في الزيادة بشكل أسرع من الدالة f(n) وذلك عندما تصبح قيمه المتغير n >= 1

ويمكننا أن تري من هذا الشكل أن ال big O notation يحد الدالة من الأعلى .. لذلك يطلق على ال big O notation بأنه upper bound asymptotic notation.

إذًا يمكن القول أن الخوارزمية السابقة تنفذ بكفاءة $O(6 * n^2)$ .. ولكن ال big o notation هو عبارة عن asymptotic notation وذلك يعني أنه يجب حذف الثوابت لذلك كفاءة هذه الخوارزمية تصبح

$O(n^2)$

– همم.. في الحقيقة نحن لسنا بحاجة للقيام بكل هذه الخطوات عند حساب الكفاءة لأي خوارزمية وفيما يلي شرح لطريقة بسيطة للقيام بذلك

حساب ال big O بشكل سهل و سريع:

الخطوات:

1. ننظر للمعادلة التي تمثل وقت تنفيذ الخوارزمية ولنسميها معادلة 1
2. نحاول إيجاد العامل المؤثر في وقت تنفيذ الخوارزمية وليكن هذا العامل A
3. تمثيل كفاءة الخوارزمية ب O(A)

فيما يلي شرح مثال لتوضيح المقصود من هذه الخطوات:

لنفترض أن وقت التنفيذ لخوارزمية مثلناه بالمعادلة التالية

$$f(n) = 5(n^2) + 3\ (n) + 2$$

بما أن n^2 هو العامل المؤثر في وقت تنفيذ الخوارزمية إذًا يمكن القول بأن هذه الخوارزمية تنفذ بكفاءة $O(n^2)$

وبذلك نعني أن وقت التنفيذ لهذه الخوارزمية لن يكون أعلى من $c * n^2$ حيث c هنا هو ثابت إذًا تم ضربه في n^2 يكون الناتج أعلى من الدالة f(n)

مثال1:

```
For (int i = 0; i < n; i++)
    c=a+b;
```

كما ذكرنا سابقًا ..الزمن الذي يستغرقه المعالج للقيام بالجمع والتخزين ثابت ويتوقف على نوع المعالج

سنرمز للزمن الذي يستغرقه معالجنا الوهمي في تنفيذ c=a+b بالثابت c1 .. و يجب أن نضع في الاعتبار أن هناك وقت زائد في تهيئة قيمة عداد الحلقة وهذا الوقت ثابت أيضًا و لنفترضه c2

و بما أن c=a+b سيتم تنفيذها داخل الحلقة n من المرات إذًا زمن تنفيذ الكود السابق يمكن تمثيله بالمعادلة التالية

$$f(n) = c1 * n + c2$$

بما أن المعادلة السابقة قيمتها تزداد بزيادة قيمة المتغير n إذًا هو العامل المؤثر في المعادلة وبحذف الثوابت يمكن القول بأن هذا الكود يتم تنفيذه بكفاءة O(n)