تحلیل زمان و فضای الگوریتم‌ها در برنامه‌نویسی

یکی از مهارت‌های کلیدی در برنامه‌نویسی حرفه‌ای، توانایی تحلیل و ارزیابی کارایی الگوریتم‌هاست. در این آموزش جامع، با مفاهیم پایه‌ای و کاربردی تحلیل زمان اجرا و حافظه مصرفی آشنا خواهید شد و یاد می‌گیرید چگونه کد خود را بهینه‌تر بنویسید.

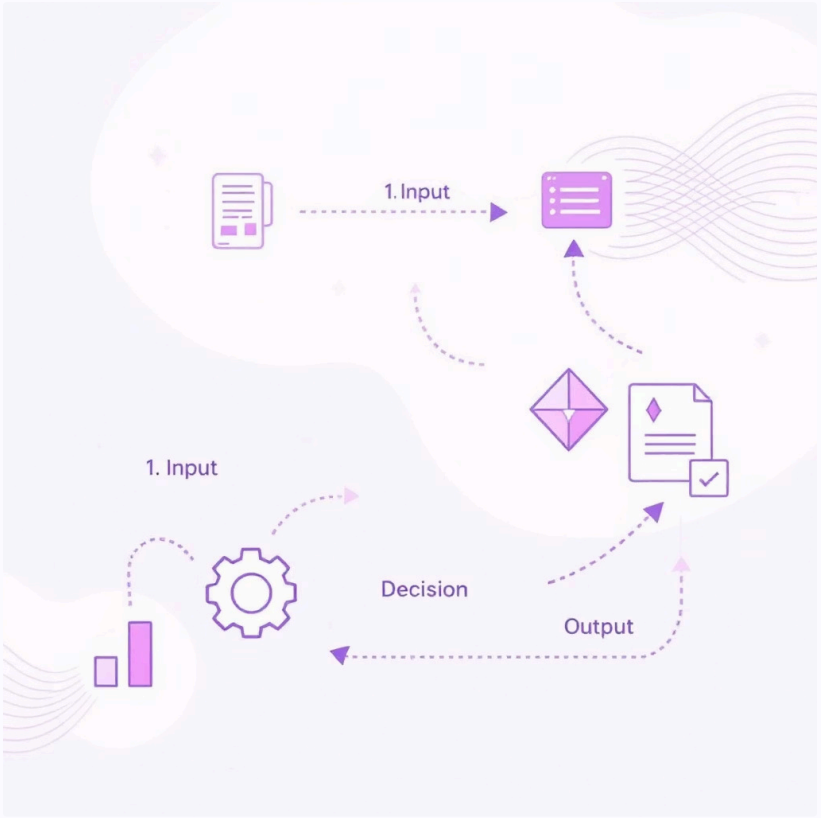
مفهوم زمان اجرای الگوریتم (Time Complexity)

🔹 تعریف دقیق و کاربردی

زمان اجرای الگوریتم به معنای **تعداد عملیات اساسی** است که الگوریتم برای حل یک مسئله انجام می‌دهد. این مفهوم مستقل از سخت‌افزار، سیستم عامل، زبان برنامه‌نویسی یا پیاده‌سازی خاص است.

نکته کلیدی: ما زمان واقعی (Real Time) را اندازه نمی‌گیریم، بلکه تعداد گام‌های محاسباتی را می‌شماریم. این رویکرد به ما امکان می‌دهد الگوریتم‌ها را به‌صورت علمی و مستقل از محیط اجرا مقایسه کنیم.

برای تحلیل علمی و دقیق الگوریتم‌ها، از نمادگذاری مجانبی (Asymptotic Notation) استفاده می‌کنیم که رفتار الگوریتم را در ورودی‌های بزرگ توصیف می‌کند.



نمادهای مجانبی: زبان تحلیل الگوریتم‌ها

Big Omega - حداقل زمان

Ω(f(n)) بهترین حالت اجرا را نشان می‌دهد. این نماد کمتر کاربرد دارد چون بهترین حالت معمولاً در دنیای واقعی کمتر اتفاق می‌افتد.

Big O - حداکثر زمان

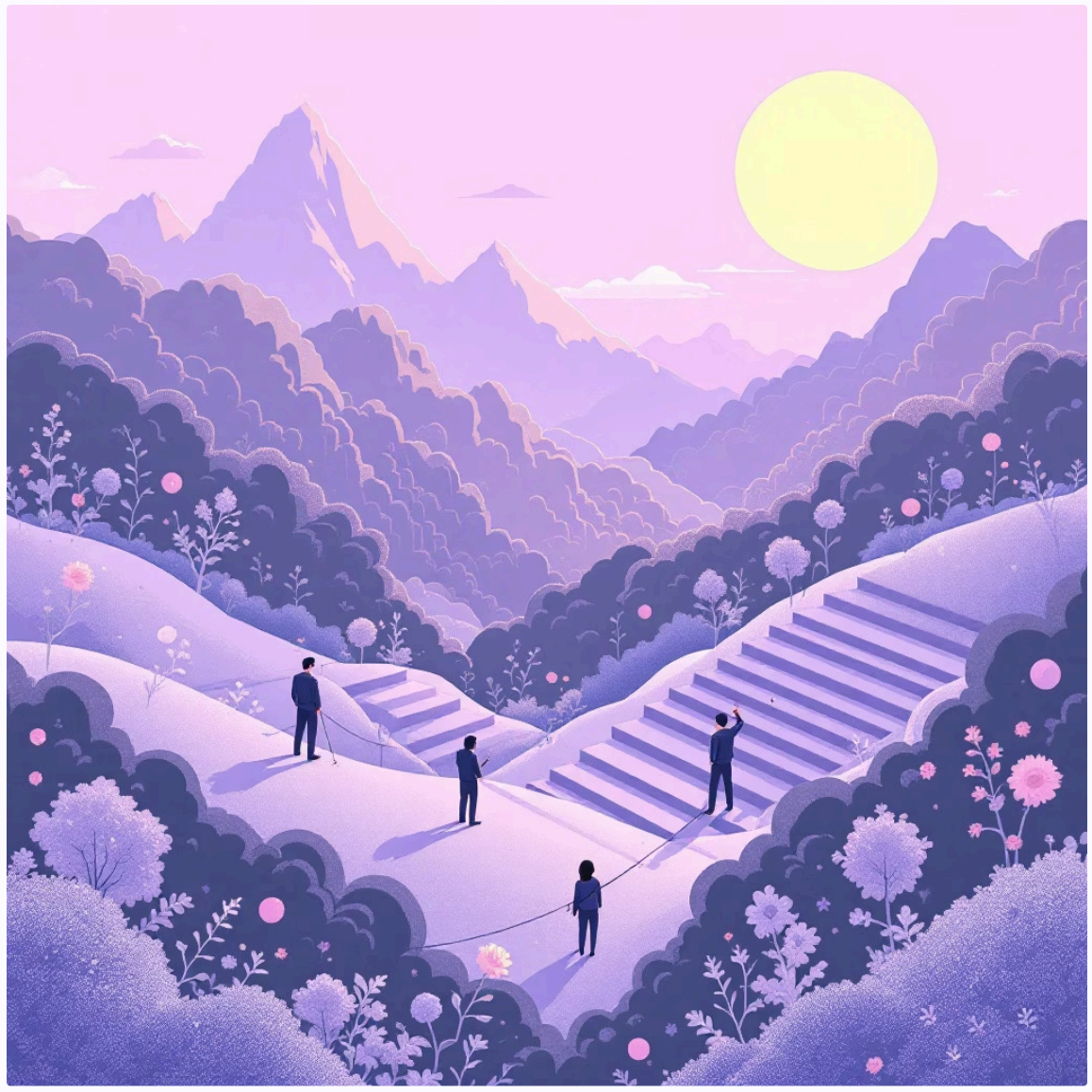
O(f(n)) بدترین حالت اجرا را نشان می‌دهد. این مهم‌ترین نماد برای تحلیل الگوریتم‌هاست چون تضمین می‌کند الگوریتم در بدترین شرایط هم از این زمان بیشتر نمی‌برد.

Big Theta - حالت متوسط

Θ(f(n)) رفتار دقیق الگوریتم را در حالت متوسط نشان می‌دهد. زمانی استفاده می‌شود که حد بالا و پایین یکسان باشند.

در این دوره، تمرکز اصلی ما روی **Big O** است زیرا در تحلیل عملی و طراحی سیستم‌های واقعی، شناخت بدترین حالت برای تضمین عملکرد و قابلیت اطمینان بسیار حیاتی است.

مثال اول: تحلیل خط‌به‌خط کد ساده



📝 کد مورد بررسی

```
a = 10
b = 20
sum = a + b
print(sum)
```

این ساده‌ترین نوع برنامه است که فقط چند عملیات انتساب و یک عملیات ریاضی دارد. بیایید به‌صورت دقیق ببینیم در پشت صحنه چه اتفاقی می‌افتد.

🔍 شمارش عملیات به تفکیک

1	مقدار 10 در حافظه ذخیره می‌شود	a = 10	1
1	مقدار 20 در حافظه ذخیره می‌شود	b = 20	2
3	خواندن a، خواندن b، جمع کردن	sum = a + b	3
1	خواندن و نمایش sum	print(sum)	4

نتیجه‌گیری

جمع کل عملیات = **6 عمل ساده**. از آنجا که تعداد عملیات کاملاً مستقل از هر ورودی متغیر است و همیشه ثابت می‌ماند، زمان اجرای این برنامه **O(1)** یا زمان ثابت (Constant Time) است.

تحلیل حافظه مصرفی (Space Complexity)

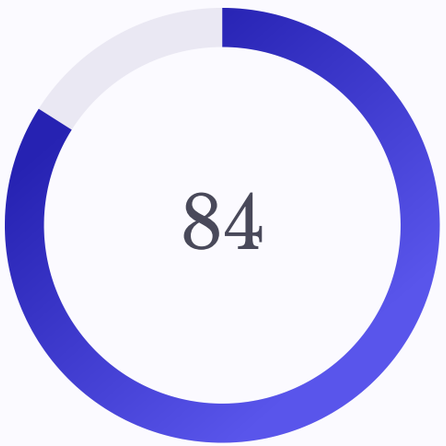
علاوه بر زمان اجرا، باید به میزان حافظه‌ای که برنامه مصرف می‌کند نیز توجه داشته باشیم. در پایتون، هر نوع داده فضای مشخصی در حافظه اشغال می‌کند.

💾 محاسبه حافظه مثال قبل

a	int	28 بایت
b	int	28 بایت
sum	int	28 بایت

📊 اندازه انواع داده در پایتون

int (عدد صحیح)	24-28 بایت
float (اعشاری)	24-28 بایت
bool (بولین)	24 بایت
str (رشته)	49+ بایت
list (لیست)	64 + 8n بایت



بایت حافظه کل

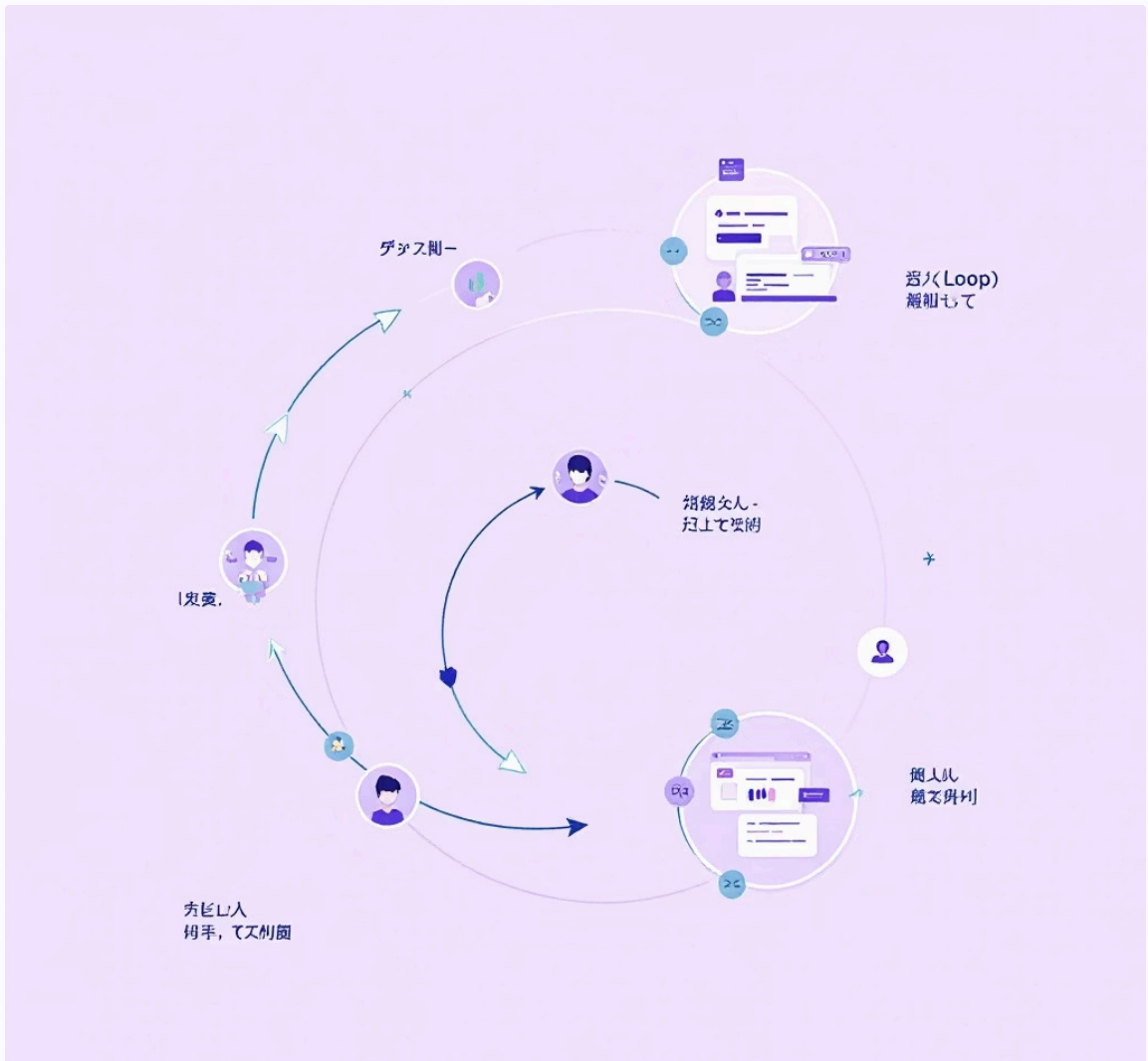
مجموع فضای مصرفی متغیرها

مستقل از ورودی

حافظه ثابت می‌ماند

چون حافظه مصرفی مستقل از اندازه ورودی است، پیچیدگی فضایی این برنامه **O(1)** است.

مثال دوم: تحلیل حلقه ساده



💻 کد با حلقه for

```
n = 5
total = 0
for i in range(1, n + 1):
    total = total + i
print(total)
```

در این مثال، یک حلقه داریم که n بار تکرار می‌شود. تعداد عملیات دیگر ثابت نیست و به اندازه ورودی بستگی دارد.

📋 تحلیل دقیق هر خط

مقداردهی اولیه	1 بار	n = 5	1
مقداردهی متغیر جمع	1 بار	total = 0	2
بار بدنه اجرا + 1 بار شرط رد n	بار n+1	for i in range...	3
در هر تکرار یک جمع	بار n	total = total + i	4
نمایش نتیجه نهایی	1 بار	print(total)	5

1

فرمول کلی

تعداد عملیات = 2n + 3

2

برای n بزرگ

ثابت‌ها بی‌اهمیت می‌شوند

3

نتیجه نهایی

زمان اجرا = O(n)

چرا ثابت‌ها حذف می‌شوند؟ در تحلیل مجانبی، وقتی n به سمت بی‌نهایت میل می‌کند، تاثیر ثابت‌ها در مقابل n ناچیز است. مثلاً برای n=1000000، تفاوت بین 2n+3 و 2n فقط 3 عملیات است که در مقابل 2 میلیون عملیات، قابل صرف‌نظر کردن است.

تحلیل حافظه مثال دوم

🧮 محاسبه دقیق فضای مصرفی

ورودی اصلی برنامه	28 بایت	int	n
متغیر جمع‌کننده	28 بایت	int	total
شمارنده حلقه	28 بایت	int	i

جمع حافظه

84 بایت (سه متغیر integer)

رابطه با n

کاملاً مستقل از ورودی

پیچیدگی فضایی

فضای ثابت - **O(1)**

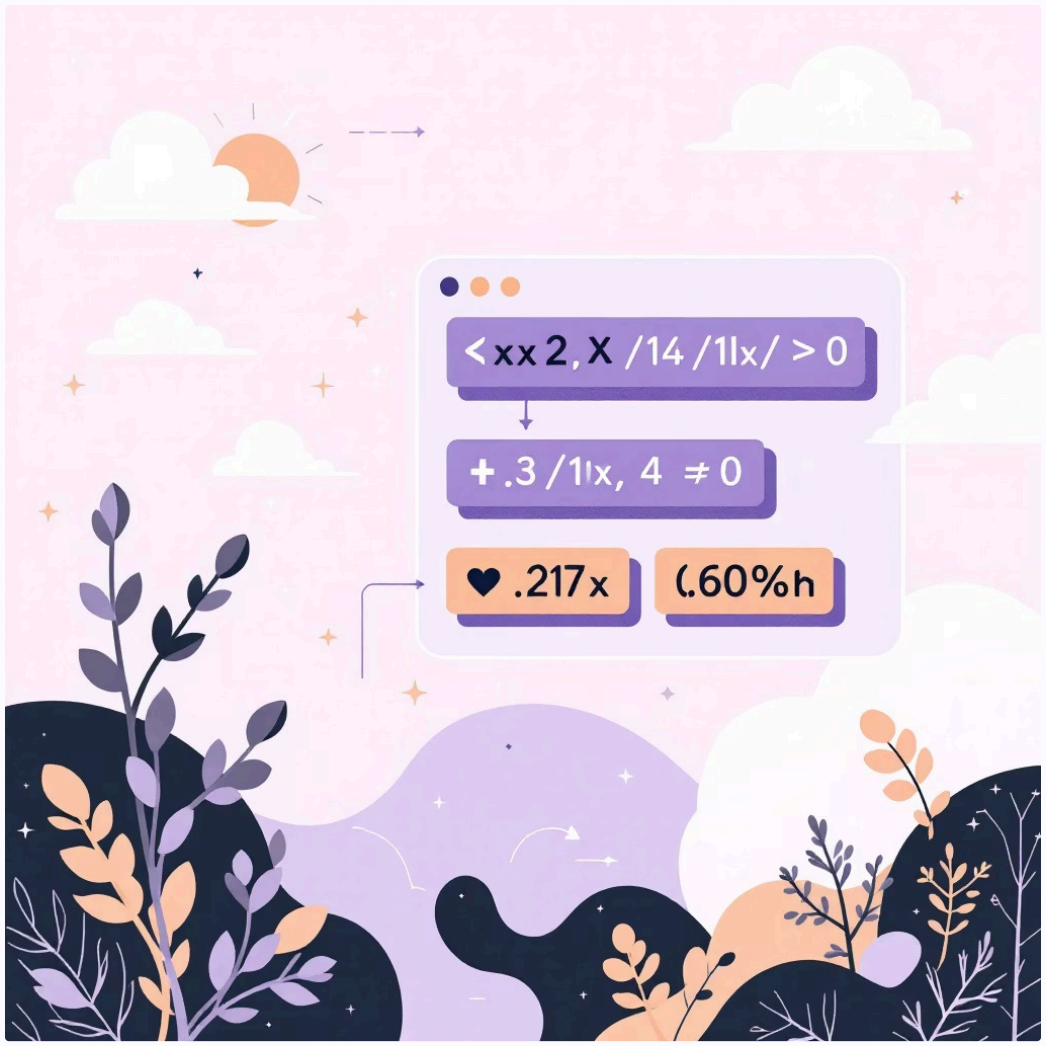
⚡ مقایسه زمان و فضا

زمان: O(n) - متناسب با ورودی رشد می‌کند
فضا: O(1) - همیشه ثابت می‌ماند
این نشان می‌دهد که یک الگوریتم می‌تواند در زمان و فضا پیچیدگی‌های متفاوتی داشته باشد.

🔑 نکته کلیدی

در این برنامه هیچ ساختار داده‌ای مثل آرایه یا لیست ایجاد نمی‌شود. فقط چند متغیر ساده داریم که فضای ثابتی اشغال می‌کنند. حتی اگر n را به 1 میلیون تغییر دهیم، باز هم همین سه متغیر وجود دارند.

مثال سوم: ذخیره‌سازی داده در لیست



📝 کد با ذخیره‌سازی

```
n = 5
numbers = []
for i in range(1, n+1):
    numbers.append(i)
print(numbers)
```

در این مثال برخلاف مثال قبل، داده‌ها را در یک لیست ذخیره می‌کنیم. این تفاوت ساده، تاثیر قابل توجهی روی حافظه مصرفی دارد.

🔍 تحلیل خط‌به‌خط دقیق

تعیین اندازه لیست مورد نظر	1 بار	n = 5	1
ایجاد لیست خالی در حافظه	1 بار	numbers = []	2
حلقه n بار اجرا + یک بار چک پایان	بار n+1	for i in range	3
افزودن هر عدد به انتهای لیست	بار n	numbers.append(i)	4
نمایش لیست کامل	1 بار	print(numbers)	5

01 محاسبه زمان اجرا

جمع عملیات ≈ 2n + 3 → **O(n)**

02 تحلیل حافظه مصرفی

لیست n عضو دارد → هر عضو 8 بایت اشاره‌گر + 64 بایت overhead

03 نتیجه فضایی

فضای کل ≈ 8n + 64 بایت → **O(n)**

تفاوت کلیدی با مثال قبل: در مثال قبل فقط یک متغیر total داشتیم که مقدار آن تغییر می‌کرد. اینجا یک لیست داریم که هر بار بزرگ‌تر می‌شود. پس هم زمان اجرا و هم فضای مصرفی O(n) هستند.

مثال چهارم: حلقه تودرتو (Nested Loop)

🔢 محاسبه تعداد اجرا

- حلقه بیرونی (i): **n بار** اجرا می‌شود
- حلقه درونی (j): برای **هر مقدار** i، j نیز **n بار** اجرا می‌شود
- جمع کل: n × n = **n^2 بار**

💻 کد با دو حلقه تودرتو

```
n = 4
for i in range(n):
    for j in range(n):
        print(i, j)
```

حلقه‌های تودرتو یکی از رایج‌ترین ساختارهایی هستند که پیچیدگی زمانی بالاتر ایجاد می‌کنند. در اینجا یک حلقه داخل حلقه دیگر قرار دارد.

i = 0

print از 0 تا 3 → 4 بار j

i = 1

print از 0 تا 3 → 4 بار j

i = 2

print از 0 تا 3 → 4 بار j

i = 3

print از 0 تا 3 → 4 بار j

16

تعداد کل اجرا

برای n=4 داریم 4×4=16

n^2

فرمول کلی

برای هر n دلخواه

💾 پیچیدگی فضایی

تنها دو متغیر i و j وجود دارند که هر کدام یک عدد صحیح هستند. مقدار این متغیرها تغییر می‌کند اما تعداد آن‌ها ثابت است.

نتیجه: O(1) - فضای ثابت

⏱️ پیچیدگی زمانی

با توجه به اینکه حلقه داخلی n بار و برای هر تکرار حلقه بیرونی که خودش n بار است اجرا می‌شود، زمان کل متناسب با n^2 رشد می‌کند.

نتیجه: $O(n^2)$ - زمان درجه دوم

درس مهم: این مثال نشان می‌دهد که زمان و فضا می‌توانند کاملاً متفاوت باشند. اینجا زمان $O(n^2)$ است اما فضا O(1). این الگو در بسیاری از الگوریتم‌های جستجو و مرتب‌سازی دیده می‌شود.

اصول شمارش دقیق و تحلیل مجانبی

🧠 عملیات پایه (Basic Operations)

در تحلیل دقیق الگوریتم‌ها، ما عملیات اساسی زیر را می‌شماریم:

انتساب

مثل: x = 5 یا result = a + b

عملیات ریاضی

جمع، تفریق، ضرب، تقسیم

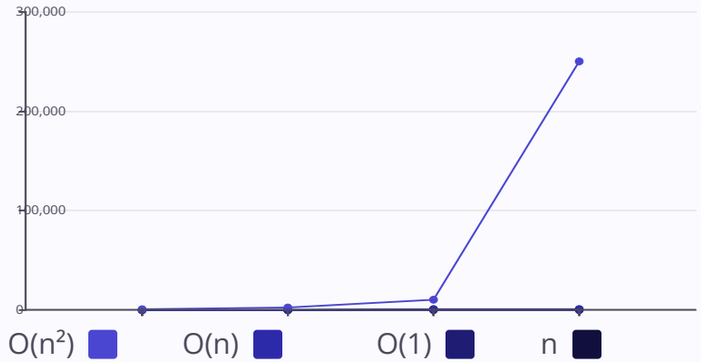
مقایسه

عملگرهای ==، >، <، !=

دسترسی به حافظه

خواندن یا نوشتن در آرایه

📈 مفهوم روند رشد (Growth Rate)



وقتی n به اعداد بزرگ می‌رسد (مثلاً میلیون‌ها رکورد)، تفاوت بین 2n و 2n+100 بسیار ناچیز است. به همین دلیل در تحلیل مجانبی:

- ضرایب ثابت را **حذف می‌کنیم**: 5n → n
- جملات کوچک‌تر را **نادیده می‌گیریم**: $n^2 + 100n + 50 \rightarrow n^2$
- فقط سریع‌ترین جمله رشد را نگه می‌داریم

🎯 نکته طلایی

تحلیل Big O به شما می‌گوید الگوریتم شما با رشد داده چطور رفتار خواهد کرد. این دانش برای طراحی سیستم‌های مقیاس‌پذیر حیاتی است.

💡 کاربرد عملی

با دانستن پیچیدگی، می‌توانید پیش‌بینی کنید آیا الگوریتم شما برای میلیون‌ها کاربر کار می‌کند یا خیر.

🚀 گام بعدی

حالا که اصول پایه را یاد گرفتید، آماده‌اید الگوریتم‌های پیچیده‌تر و تکنیک‌های بهینه‌سازی را بیاموزید!