طراحی الگوریتم

موسسه آموزش عالی پاسارگاد شیراز

مقدمه

- منابع درس
  - طراحی الگوریتم کورمن (CLRS)
  - طراحی الگوریتم هادی یوسفی
  - ساختمان داده (مدرسان شریف)

- مباحث درس
  - مرتبه اجرایی
  - روش تقسیم و غلبه
  - برنامه نویسی پویا
  - روش حریصانه
  - روش عقبگرد

تحلیل پیچیدگی زمانی تعیین گام‌های یک برنامه

یک گام از برنامه (program step) از نظر دستوری یا معنایی، قسمت با معنی یک برنامه است که دارای زمان اجرایی مستقل از مشخصه‌های موردی است. تعداد گام‌های هر دستور از برنامه، به ماهیت آن دستور باز می‌گردد. در این بحث سعی ما بر این است که انواع مختلفی از دستور را که در هر برنامه می‌تواند وجود داشته باشد، بررسی کنیم.

۱ـ عبارات توضیحی (Comments) : توضیحات، عبارات غیراجرایی هستند و تعداد گام‌های آن صفر است.

۲ـ عبارت تعیین نوع (Declarative statements) : این طبقه شامل عباراتی نظیر معرفی متغیرها و ثابت‌ها و انواع جدید و... می‌باشد. تعداد گام‌های این عبارات صفر است، چون اجرایی نیستند.

۳ـ عبارات و دستورات انتساب (Expressions and assignment statements) : تعداد تکرار بیشتر دستورات، یک است. به جز عباراتی که شامل فراخوانی توابع می‌باشد. در این حالت باید هزینهٔ لازم برای اجرای توابع را محاسبه کنیم.

۴ـ عبارات تکرار (Iteration statements) : این دسته از عبارات، شامل عبارت for، while است. شمارش گام را فقط برای قسمت کنترل این عبارت در نظر می‌گیریم.

به طور کلی حلقه‌های تکرار به دو دسته تقسیم می‌شوند:

الف) حلقه‌های تکراری که ابتدا شرط را بررسی کرده و سپس دستورات را اجرا می‌کنند؛ در این مورد اگر تعداد تکرار دستورات k بار باشد، تعداد تکرار قسمت کنترل حلقه $k+1$ بار خواهد بود.

ب) حلقه‌های تکراری که ابتدا دستورات را اجرا کرده و در آخر شرط را بررسی می‌کنند؛ در این مورد اگر تعداد تکرار دستورات k بار باشد، تعداد تکرار قسمت کنترل حلقه نیز k بار خواهد بود.

۵ـ عبارات { و} : تعداد گام هر عبارت {و} صفر است.

۶ـ عبارت نام تابع: تعداد گام این عبارت صفر است.

۷ـ عبارت return: تعداد گام این عبارت یک است.

با انتساب مقادیر فوق به شمارش گام عبارات، می‌توان تعداد گام‌های یک برنامهٔ خاص را تعیین کرد. برای تعیین شمارش گام برنامه باید کل تعداد گام‌های هر عبارات را لیست کنیم. در این حالت، ابتدا تعداد گام‌ها را در هر اجرای عبارت و تعداد کل دفعات اجرای هر عبارت را تعیین می‌کنیم. با ترکیب این دو مقدار، کل زمان اجرای هر عبارت به دست می‌آید. با جمع کردن این مقادیر برای همهٔ عبارات، شمارش گام برنامه به دست خواهد آمد.

```
void main()          →0
{                    →0
    float b=0;       →1
    a=12;            →1        ⟹ 3
    b=a+b;           →1
}                    →0
```

```
void main()                          →0
{                                    →0
    int a=5;                         →1
    int i=0;                         →1      ⟹ 13
    for(int j=0;j<a;j++)             →6
        i=i+1;                       →5
}                                    →0
```

```
void main()          → 0
{                    → 0
    int n;           → 0
    cin >> n;        → 1
    int i=0;         → 1        ⇒ 2n+3
    for(int j=0; j<n; j++)  → n+1
        i=i+1;       → n
}                    → 0
```

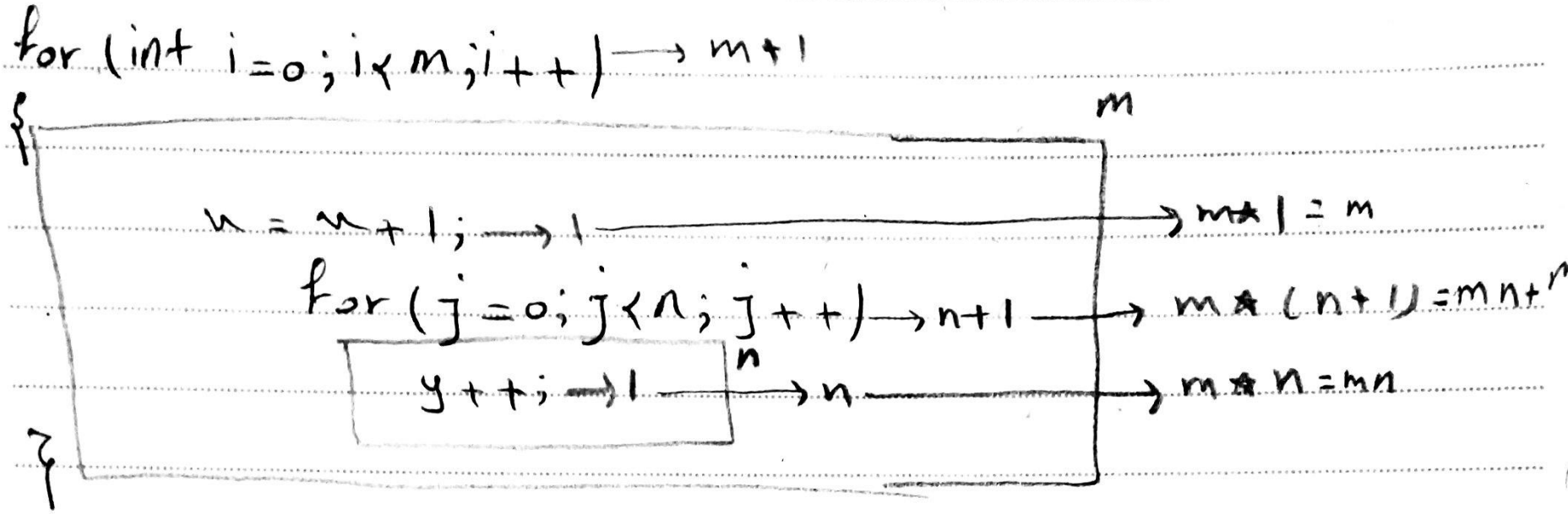
نکته: برای محاسبه پیچیدگی مشخص می‌کنیم دستورات اصلی چند بار تکرار می‌شود (براساس ورودی)

```
void f1(int m, int n)
{
    int a=0;
    for(int i=0; i<n; i++)
        for(int j=0; j<m; j++)
            a++;            → n * m
}
```

```
for (int i=1; i<=n; i*2)
    a++;     → log_2^n
```

* اگر در حلقه‌ای شمارنده با ضریبی افزایش پیدا کند تعداد تکرار از مرتبه n log در مبنای ضریب است.

مثال: / تعداد عملیات مقابل چقدر است؟!



$$m+1+m(1+(n+1)+n) = 2mn+3m+1$$

بررسی پیچیدگی الگوریتم در بهترین، بدترین و حالت میانگین:

مثال:

```
for(int i=0; i<n; i++)
    k++;
```

1) بدترین حالت: $n+1+n$

2) بهترین حالت: $n+1+n$

3) حالت میانگین: $n+1+n$

مثال: حالت‌های مختلف را برای الگوریتم جستجوی یک آرایه یک‌بعدی به تعداد n عضو را بررسی کنید.

```
int search (int list[], int x, int n)
{
    for (int i=0; i<n ; i++)
    {
        if (list[i]==x )
            return i; // عنصر در خانه iام است
    }
    return -1; // عنصر نباشد
}
```

بهترین حالت: از مرتبه 1

حالت متوسط:

$$\sum \frac{1}{n} i = \frac{1}{n} \sum i = \frac{1}{n} \frac{n(n+1)}{2} =$$

$$= \frac{n+1}{2}$$

بدترین حالت: n

انگیزه‌ی ما برای تعیین شمار گام‌ها و تعداد عملیات الگوریتم برای اندازه‌ی ورودی، توانایی مقایسه پیچیدگی‌های زمانی دو برنامه‌ای است که یک عمل را انجام می‌دهند و نیز پیش‌بینی رشد زمان اجرا با تغییر مشخصه‌های موردی است.

در عمل، محاسبات دقیق فوق و یا تعیین شمار دقیق گام‌های (بدترین حالت یا حالت میانگین) یک برنامه، کار بسیار مشکلی است.

تلاش و کوشش زیاد برای تعیین دقیق شمار گام‌ها، مقرون به صرفه نیست. از طرف دیگر، هدف از تحلیل یک الگوریتم، به دست آوردن تقریبی از رابطه‌ی آن است که محاسبه آن ساده باشد و همچنین این تقریب بیانگر میزان کارایی الگوریتم بوده و شامل اطلاعات اضافی نباشد.

الگوریتم‌ها اغلب براساس نرخ رشد تابع تعداد عملیات، دسته‌بندی و مقایسه می‌شوند و زمانی که ورودی تابع به اندازه کافی بزرگ باشد، جملاتی که در بزرگ شدن اندازه‌ی تابع نقش زیادی ندارند قابل چشم پوشی بوده و حذف می‌گردند. به عبارت دیگر، پیچیدگی یک الگوریتم به ازای مقادیر به اندازه کافی بزرگ ورودی، به مقدار واقعی تعداد عملیات انجام شده در الگوریتم به ازای آن ورودی نزدیک است. به مثال زیر توجه کنید:

مثال : کدام یک از جملات رابطه زیر، به ازای مقادیر بزرگ ورودی، بیشترین نقش را در رشد تابع دارد؟

$$\mathbf{t(n) = n^3 + 50n\ \log n + 1000n + 5000}$$

اگر مقدار تابع t را به ازای مقادیر بزرگ ورودی n حساب کنیم، n^3 بزرگ‌ترین جمله بوده و به دلیل نرخ رشد بیشتری که نسبت به سه جمله دیگر دارد، بزرگ‌ترین باقی می‌ماند. به عنوان نمونه، هنگامی که $n = 100$ شود، جمله اول در مقایسه با سه جمله دیگر، به حدی بزرگ می‌شود که می‌توان از سایر جملات تابع صرف‌نظر کرد و بنابراین رشد تابع t تقریباً وابسته به رشد جمله n^3 است.

نماد O :

نماد مجانبی O (Big-O) ، حد بالای نرخ رشد توابع را نشان می‌دهد. $f(n) \in O(g(n))$ (خوانده می‌شود f ،O,n بزرگ g ی n است) به این معنی است که نرخ رشد f، کوچک‌تر یا مساوی نرخ رشد تابع g است. تعریف ریاضی آن به صورت زیر می‌باشد:

$$f(n) \in O(g(n)) \longleftrightarrow \{\exists\, c > 0, n_\circ > 0 \mid \forall\, n \geq n_\circ \;\; ; \;\; f(n) \leq c.\, g(n)\}$$

طبق این تعریف تابع f متعلق به مجموعه $O(g(n))$ است، اگر و فقط اگر، ثابت‌های مثبتی مانند $n_\circ, c$ وجود داشته باشد به طوری که به ازای تمام nهای به اندازه کافی بزرگ $(n \geq n_\circ)$، مقدار $f(n)$ کوچک تر یا مساوی تابع $c.\, g(n)$ باشد.

می‌توان نشان داد که $5n^2 + 3n + 6 \in O(n^2)$ و یا $6n + 5 \in O(n^2)$ و به طور کلی $O(n^2)$ شامل همه توابعی است که نرخ رشد آن‌ها کمتر یا مساوی n^2 است.

تذکر۱: در برخی موارد $f(n) \in O(g(n))$ را به صورت $f(n) = O(g(n))$ نیز نمایش می‌دهند.

نماد Ω :

نماد مجانبی Ω (Big-Omega) ، حد پایین نرخ رشد توابع را نشان می‌دهد. $f(n) \in \Omega(g(n))$ ، (خوانده می‌شود f ، n ، اُمگای g ی n است) به این معنی است که رشد تابع f بزرگ‌تر یا مساوی نرخ رشد تابع g است. تعریف ریاضی آن به صورت زیر است:

$$f(n) \in \Omega(g(n)) \leftrightarrow \{\, \exists\, c > 0, n_\circ > 0 \mid \forall\, n \geq n_\circ \;;\; f(n) \geq c.g(n) \,\}$$

طبق این تعریف تابع f متعلق به مجموعه $\Omega(g(n))$ است، اگر و فقط اگر ثابت‌های مثبتی مانند $n_\circ, C$ وجود داشته باشد به طوری که به ازای n های به اندازه کافی بزرگ $(n \geq n_\circ)$ ، مقدار تابع $f(n)$ ، بزرگ‌تر یا مساوی $c.g(n)$ باشد.

می‌توان نشان داد که $7n^2 + n + 5 \in \Omega(n^2)$,$6n^4 + 2n^2 + 1 \in \Omega(n^2)$ و به طور کلی $\Omega(n^2)$ شامل همه توابعی است که رشد آن‌ها بیشتر یا مساوی n^2 است.

نماد θ:

نماد مجانبی θ، زمانی استفاده می‌شود که نرخ رشد دو تابع مساوی باشد. تعریف ریاضی آن به صورت زیر است:

$$f(n)\in\theta(g(n))\longleftrightarrow\{\exists\, c_1,c_2,n_\circ>0\,|\forall n>n_\circ;c_1.g(n)\le f(n)\le c_2.g(n)\}$$

طبق این تعریف تابع f متعلق به $\theta(g(n))$ است، اگر و فقط اگر ثابت‌های مثبتی مانند $n_\circ,c_2,c_1$ وجود داشته باشند، به طوری که به ازای nهای به اندازه کافی بزرگ $(n\ge n_\circ):c_1.g(n)\le f(n)\le c_2.g(n)$ باشد.

می‌توان نشان داد که $3n+3\in\theta(n)$ و یا $10n^2+4n+2\in\theta(n^2)$ و $10n^2+4n+2\notin\theta(n)$.

نکته: θ دقیق‌ترین بیان برای نرخ رشد یک تابع است.

نماد مجانبی o (small-o) همانند نماد $Big-O$ است با این تفاوت که حد بالای اکید (مطلق) را برای رشد توابع مشخص می‌کند. $f(n)\in o(g(n))$ به این معنی است که رشد $f(n)$ همیشه کوچک‌تر از رشد تابع $g(n)$ خواهد بود.

تعریف ریاضی آن به صورت زیر است :

$$f(n)\in o(g(n))\longleftrightarrow\{\forall c>0\ ,\exists n_\circ>\circ|\forall\ n>n_\circ\ ;\ 0\le f(n)<c.g(n)\}$$

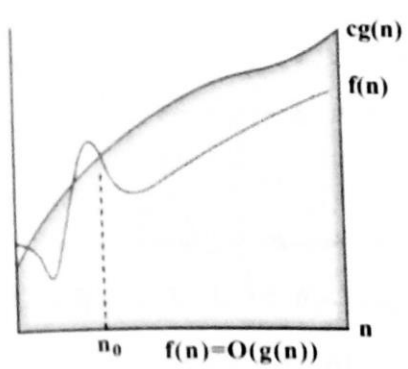
نماد ω:

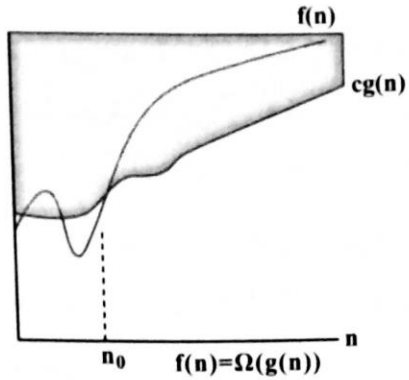
نماد مجانبی ω (small – omega) نیز همانند نماد $Big-Omega$ است و حد پایینی اکید (مطلق) را برای رشد تابع ارائه می‌دهد. $f(n)\in\omega(g(n))$ به این معنی است که رشد f(n) ، همیشه بیشتر از رشد تابع g(n) است.

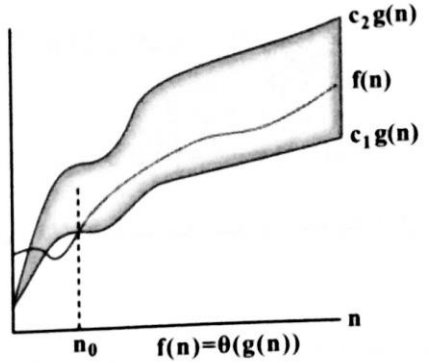
تعریف ریاضی آن به صورت زیر است :

$$f(n)\in\omega(g(n))\longleftrightarrow\{\forall c>0,\exists n_\circ>\circ|\forall n>n_\circ\ ;\ 0\le c.g(n)<f(n)\}$$

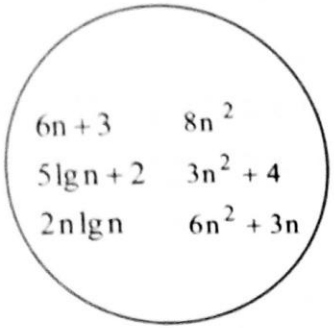
نمودارهای زیر مقایسه‌ای از تعاریف θ, Ω, O را نشان می‌دهند:



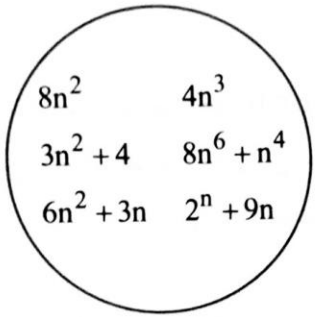




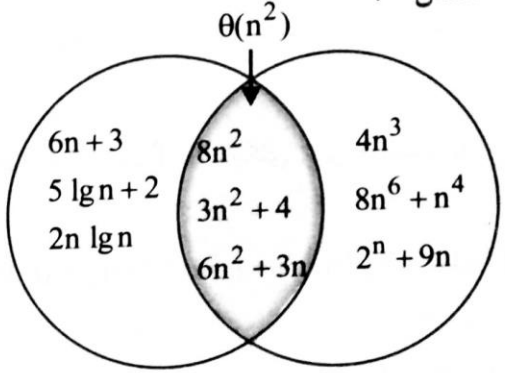
مثال :



$O(n^2)$



$\Omega(n^2)$



$\theta(n^2)=O(n^2)\cap\Omega(n^2)$

$O(1) < O(\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$

مثال:
```
for (int i=0; i<1000; i++)
    k++;
```
$f(n) = O(1)$

مثال:
```
for (int i=0; i<n; i++)
    for (int j=0; j<i; j++)
        k++;
```
$$\sum_{i=0}^{n} \sum_{j=0}^{i} 1 = \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

مثال:
```
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        n++;
```
$O(n^2)$

مثال:
```
for (int i=n; i>=1; i/=2)
```
$O(\log_2^n)$

مثال:
```
for (int i=1; i<n; i*=3)
    for (int j=0; j<1000; j++)
        n++;
```
$O(\log_3^n)$ ← $O(1)$

نکته: اگر مرتبه اجرای الگوریتمی برابر با $O(n_1)$ باشد و زمان اجرای آن در کامپیوتری با سرعت v_1 برابر با t_1 باشد از طرفی مرتبه اجرای الگوریتمی دیگر برابر با $O(n_2)$ باشد و زمان اجرای آن در کامپیوتری دیگر با سرعت v_2 برابر با t_2 باشد آنگاه:

$$\frac{O(n_2)}{O(n_1)} = \frac{v_2 * t_2}{v_1 * t_1}$$

مثال ۱۰: اگر زمان اجرای الگوریتمی با پیچیدگی $n^2 \log n$ برای $n=10$ برابر

۲ (ثانیه) باشد زمان اجرای همین الگوریتم بر روی این کامپیوتر برای $n=100$ چقدر

خواهد بود؟!

فرض میکنیم $\log 100 = 2$ و $\log 10 = 1$

$$\frac{o(n_2)}{o(n_1)} = \frac{v_2 * t_2}{v_1 * t_1}$$

$$\frac{n_2^2 \log n_2}{n_1^2 \log n_1} = \frac{v_2 * t_2}{v_1 * t_1}$$

$n_1 = 10$

$n_2 = 100$

$o(n_1) = n^2 \log n$

$o(n_2) = n^2 \log n$

$v_1 = v_2$

$t_1 = 2\,ns$

$t_2 = ?$

$$\frac{(100)^2 \log 100}{(10)^2 \log 10} = \frac{t_2}{2} \Rightarrow$$

$$\frac{10000 \times 2}{100 \times 1} = \frac{t_2}{2} \Rightarrow t_2 = 400\,ns$$