دانشکده‌ی علوم ریاضی

آنالیز الگوریتم‌ها ۱۷ بهمن ۱۳۹۱

# جلسه‌ی دوم: تحلیل مجانبی، قضیه اصلی

*مدرّس: دکتر شهرام خزائی* *نگارندگان: جواد عابدی‌گزل‌آباد*

## ۱ مقدمه

در حل بسیاری از مسائل تئوری به جای اثبات، از طراحی الگوریتم استفاده می‌شود. درنتیجه برای ارزیابی راه‌حل بهینه نیاز به معیاری برای مقایسه الگوریتم‌های مختلف داریم. در این جلسه قصد داریم ابزارهای مناسب برای اینکار را بدست آوریم. سپس به بیان چند مثال در این ارتباط می‌پردازیم.

## ۲ ابزار لازم برای تحلیل مجانبی الگوریتم‌ها

**تعریف ۱** *می‌گوییم $T(n) = O(f(n))$ اگر و فقط اگر ثابت‌های $c$ و $n_0$ وجود داشته باشند که به ازای هر $n \geq n_0$ داشته باشیم $T(n) \leq cf(n)$.*

**قضیه ۱** *اگر $T(n) = a_k n^k + a_{k-1} n^{k-1} + ... + a_0$ آنگاه $T(n) = O(n^k)$.*

*برهان.* در صورتیکه ثابت‌ها به شکل زیر مشخص گردند:

$$n_0 = 1$$
$$c = |a_k| + |a_{k-1}| + \cdots + |a_0|$$

به وضوح شرط مسئله برقرار خواهد بود. ■

**قضیه ۲** *اگر $T(n) = n^k$ آنگاه $T(n) \neq O(n^{k-1})$.*

*برهان.* برهان خلف: فرض کنید $T(n) = O(n^{k-1})$ باشد. لذا ثابت‌های $c$ و $n_0$ وجود دارند بطوری که به ازای هر $n \geq n_0$ داریم:

$$cn^{k-1} \geq n^k$$

در اینصورت نتیجه می‌شود:

$$c \geq n$$

که نتیجه‌ای اشتباه است. درنتیجه حکم اصلی مسئله برقرار خواهد بود. ■

**مسأله ۱** *اگر داشته باشیم* $T(n) = ۲^{n+۱۰}$ *، آیا* $T(n) = O(۲^n)$ *است؟*

**جواب ۱** بله. *با انتخاب مناسب ضرایب به شیوه‌ی زیر حکم اثبات می‌شود:*

$$c = \frac{۲^{n+۱۰}}{۲^n} = ۲^{۱۰}$$
$$n_۰ = ۱$$

**مسأله ۲** *اگر داشته باشیم* $T(n) = ۲^{۱۰n}$ *، آیا* $T(n) = O(۲^n)$ *است؟*

**جواب ۲** خیر. برهان خلف: *فرض کنید* $T(n) = O(۲^n)$ *باشد. لذا ثابت‌های* $c$ *و* $n_۰$ *وجود دارند بطوری که به ازای هر* $n \geq n_۰$ *داریم:*

$$c۲^n \geq ۲^{۱۰n}$$

در اینصورت نتیجه می‌شود که:

$$c \geq ۲^{۹n}$$

که نتیجه‌ای اشتباه است. درنتیجه حکم مسئله برقرار نیست.

**تعریف ۲** *می‌گوییم* $T(n) = \Omega(f(n))$ *اگر و فقط اگر ثابت‌های* $c$ *و* $n_۰$ *وجود داشته باشند که به ازای هر* $n \geq n_۰$ *داشته باشیم:* $T(n) \geq cf(n)$.

**مثال ۱** *در نتیجه با این تعریف داریم:*

$$n^k = \Omega(n^{k-۱})$$
$$۲^{۱۰n} = \Omega(۲^n)$$

**تعریف ۳** *می‌گوییم* $T(n) = \theta(f(n))$ *اگر و فقط اگر* $T(n) = \Omega(f(n))$ *و* $T(n) = O(f(n))$.

نمادهایی که هم‌اکنون برای تحلیل الگوریتم‌ها استفاده می‌شوند توسط Donald Ervin Knuth در سال ۱۹۷۶ پیشنهاد شده و مورد قبول همگان قرار گرفته است. Knuth را پدر آنالیز الگوریتم‌ها نیز می‌نامند.

## ۳ تکنیک تقسیم و حل

این مفهوم در حل بسیاری از مسائل کاربرد دارد. در این روش برای حل مسئله دو مرحله زیر باید انجام شود:

- تقسیم مسئله به تعدادی زیرمسئله با اندازه کوچکتر
- حل و ادغام زیرمسئله‌ها برای حل مسئله اصلی

با فرض اینکه تعداد زیرمسئله‌ها $a$ باشد، اندازه‌ی زیرمسئله‌ها با ضریب $b$ کاهش یابد و پیچیدگی ادغام $O(n^d)$ باشد برای تحلیل الگوریتم داریم:

$$T(n) \leq aT(\frac{n}{b}) + O(n^d)$$

به عنوان مثال برای الگوریتم مرتب‌سازی ادغامی رابطه‌ی زیر برقرار است:

$$T(n) \leq ۲T(\frac{n}{۲}) + O(n)$$

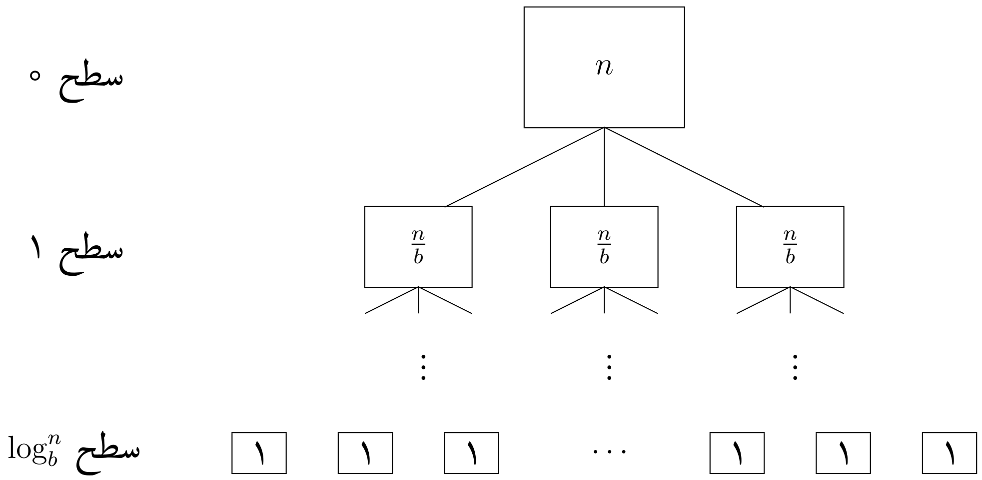
**قضیه ۳** *(قضیه اصلی): طبق تعاریف بالا اگر رابطه‌ی بازگشتی زیر را داشته باشیم:*

$$T(n) \leq aT(\frac{n}{b}) + O(n^d)$$

*آنگاه:*

$$T(n) = \begin{cases} O(n^d \log n) & if \quad a < b^d \\ O(n^d) & if \quad a = b^d \\ O(n^{\log_b^a}) & if \quad a > b^d \end{cases}$$

**برهان.** می‌توان درخت زیر را برای تصور حجم محاسبه در نظر گرفت:




شکل ۱: درخت محاسبه با فرض $a = ۳$

فرض کنید مسئله‌ی اولیه در سطح صفر باشد. پس از هر مرحله، مسئله به $a$ زیرمسئله تقسیم شده و وارد سطح بالاتر می‌شویم و هزینه تقسیم و ادغام زیرمسئله‌ها را نیز می‌پردازیم. برای محاسبه‌ی هزینه در سطح $j$ تعداد زیرمسئله‌ها را در هزینه تقسیم و ادغام هرکدام ضرب می‌کنیم. // تعداد زیرمسئله‌ها در سطح $j$ برابر است با: $a^j$
اندازه‌ی زیرمسئله‌ها در سطح $j$ برابر است با: $\frac{n}{b^j}$
فرض کنید برای $n$ های به اندازه‌ی کافی بزرگ $O(n^d) \leq c_\circ n^d$ و قرار دهید $c = \max(c_\circ, T(۱))$.
هزینه‌ی کل در این سطح حداکثر برابر است با:

$$a^j \times c(\frac{n}{b^j})^d = cn^d(\frac{a}{b^d})^j$$

در نتیجه هزینه‌ی کل مسئله برابر است با:

$$\sum_{j=\circ}^{\log_b^n} cn^d(\frac{a}{b^d})^j = cn^d \sum_{j=\circ}^{\log_b^n}(\frac{a}{b^d})^j$$

در نتیجه با توجه به اینکه کسر داخل سیگما، کمتر، مساوی یا بیشتر از یک باشد یکی از حالت‌های فرض مسئله بوجود می‌آید. ■

**مسأله ۳** *(مسئله‌ی زوج‌های معکوس[۱]) می‌خواهیم در یک آرایه‌ی $n$ عضوی تعداد زوج‌های معکوس را بیابیم.*
*تعریف. زوج عدد $(a_i, a_j)$ در آرایه‌ی $a_۱, \ldots, a_n$ را یک **زوج معکوس** می‌نامیم **اگر و تنها اگر** $i < j$ و $a_i > a_j$ باشد.*

[۱] Inversion Number

**جواب ۳** الگوریتم بدیهی: تمام زوج‌های ممکن را مورد بررسی قرار می‌دهیم و درصورتی که عضو با اندیس کمتر بیشتر از عضو با اندیس بیشتر باشد یک واحد به شمارنده‌ی خود اضافه می‌کنیم.
تعداد عملیات این الگوریتم به وضوح $O(n^۲)$ است.

الگوریتم بهتر: با روش تقسیم و حل. ابتدا آرایه را به دو بخش راست و چپ تقسیم کرده و مقدار تابع برای هر بخش را محاسبه می‌کنیم. در نهایت باید تعداد زوج‌های معکوس بین این دو بخش را نیز بیابیم.

**Algorithm 1** CountInv

```
function CountInv(A, n)
    [assumes n is a power of two]
    A_L ← left half of A
    A_R ← right half of A
    l ← CountInv(A_L, n/2)
    r ← CountInv(A_R, n/2)
    s ← SplitInv(A_L, A_R, n/2)
    A ← Merge(A_L, A_R, n/2)
    return l + r + s
```

```
function Merge(arrays A, B, size m)
    [assumes A[i] = B[i] = 0 for i > m]
    i ← 1, j ← 1
    for k = 1 to m do
        if A[i] < B[j] then
            C[k] ← A[i]
            i ← i + 1
        else
            C[k] ← B[j]
            j ← j + 1
    return C
```

از روند Merge استفاده شده است تا به کمک آن آرایه اولیه مرتب شود. در نتیجه به روش بازگشتی می‌توانیم فرض کنیم که $A_L$ و $A_R$ هر دو مرتب شده هستند. در صورتیکه چنین فرضی نداشتیم برای پیاده‌سازی روند SplitInv باید به ازای هر جفت از عناصر دو آرایه بررسی صورت می‌گرفت همانند زیر:

**Algorithm 2** SplitInv ($O(m^2)$-implementation)

```
function SplitInv(A, B, m)
    s ← 0
    for i = 1 to m do
        for j = 1 to m do
            if A[i] > B[j] then
                s ← s + 1
    return s
```

در اینصورت رابطه‌ی بازگشتی زیر را برای محاسبه زمان اجرای الگوریتم خواهیم داشت:

$$T(n) = ۲T(\tfrac{n}{۲}) + O(n^۲) = O(n^۲)$$

ولی اگر فرض مرتب بودن را نیز در طول اثبات در نظر بگیریم می‌توان تابع نهایی را بدین‌گونه تغییر داد:

**Algorithm 3** SplitInv ($O(m)$-implementation)

```
function SplitInv(A, B, m)
    s ← 0
    i ← 1
    j ← 1
    while i < m  &  j < m do
        if A[i] ≥ B[j] then
            j ← j + 1
        else
            i ← i + 1
            s ← s + (m − j + 1)
```

اندیس $j$ را که روی $B$ حرکت می‌کند تا شرط وقتی که شرط $A[i] \geq B[j]$ برقرار است یک واحد یک واحد افزایش می‌دهیم. در صورتی‌که شرط دوم ( $A[i] < B[j]$ ) برقرار شود، $A[i]$ از تمامی اعداد $B[j], \ldots, B[m]$ که تعداد آنها $m - j + ۱$ است بزرگتر است و درنتیجه مقدار $s$ را افزایش می‌دهیم.

در هر گام از این تابع پس از مقایسه‌ی $A[i]$ و $B[j]$ به یکی از دو اندیس $i$ یا $j$ یک واحد اضافه خواهد شد. در نتیجه اجرای این تابع حداکثر $O(m)$ مرحله طول می‌کشد. از طرفی می‌دانیم که زمان اجرای تابع $\textsc{Merge}(A_L, A_R, \frac{n}{۲})$ از مرتبه $O(n)$ است. پس رابطه‌ی بازگشتی زیر را برای زمان اجرای نهایی مسئله خواهیم داشت:

$$T(n) = ۲T(\tfrac{n}{۲}) + O(n) = O(n \log n)$$

**مسأله ۴** مسئله‌ی ضرب اعداد. می‌خواهیم دو عدد $n$ رقمی را در یکدیگر ضرب کنیم. الگوریتمی ارائه دهید تا به روشی سریع این کار را انجام دهد.

**جواب ۴** الگوریتم بدیهی. هر دو رقم را در یکدیگر ضرب کرده و با در نظر گرفتن جایگاه هر عدد آنها را با یکدیگر جمع می‌کنیم. در اینصورت مرتبه‌ی الگوریتم $O(n^۲)$ خواهد بود.

الگوریتم بهتر: در صورتیکه از الگوریتم تقسیم و حل استفاده کنیم. ابتدا هر عدد را به دو زیرعدد تقسیم می‌کنیم و پس از محاسبه ضرب هر دو جفت زیرعدد، چهار عدد نهایی را با ضرایب مناسب با یکدیگر جمع خواهیم کرد.

**Algorithm 4** MULTIPLY ($O(n^2)$-implementation)

```
function MULTIPLY(X, Y)
    X_L ← left half bits of X
    X_R ← right half bits of X
    Y_L ← left half bits of Y
    Y_R ← right half bits of Y
    P_1 ← MULTIPLY(X_L, Y_L)
    P_2 ← MULTIPLY(X_L, Y_R)
    P_3 ← MULTIPLY(X_R, Y_L)
    P_4 ← MULTIPLY(X_R, Y_R)
    return P_1 × 2^n + (P_2 + P_3) × 2^(n/2) + P_4
```

در نتیجه رابطه‌ی بازگشتی‌ای به شکل زیر خواهیم داشت:

$$T(n) = ۴T(\frac{n}{۲}) + O(n) \Rightarrow T(n) = O(n^{۲})$$

برای اینکه بتوانیم الگوریتم خود را بهینه کنیم، از یک ایده استفاده می‌کنیم تا یکی از حاصلضرب‌ها را از بقیه نتایج بدست بیاوریم. این ایده اولین بار توسط گوس [۲] برای محاسبه‌ی حاصلضرب اعداد مختلط استفاده شد و بعدا توسط کاراتسوبا [۳] برای حل این مساله مطرح گردید.

**Algorithm 5** MULTIPLY ($O(n^{1.59})$-implementation)

```
function MULTIPLY(X, Y)
    X_L ← left half bits of X
    X_R ← right half bits of X
    Y_L ← left half bits of Y
    Y_R ← right half bits of Y
    P_1 ← MULTIPLY(X_L, Y_L)
    P_2 ← MULTIPLY(X_R, Y_R)
    P_3 ← MULTIPLY(X_L + X_R, Y_L + Y_R)
    return P_1 × 2^n + (P_3 − P_1 − P_2) × 2^(n/2) + P_2
```

در نتیجه رابطه‌ی بازگشتی‌ای به شکل زیر خواهیم داشت:
با استفاده از این روش در هر مرحله به جای انجام ۴ ضرب، از ۳ ضرب استفاده می‌کنیم به صورتی که به همان جواب اصلی برسیم. حال پس از این بهینه‌سازی رابطه‌ی بازگشتی به شکل زیر خواهد رسید:

$$T(n) = ۳T(\frac{n}{۲}) + O(n) \Rightarrow T(n) = O(n^{\log_۲^۳}) = O(n^{۱/۵۹})$$

[۲] Carl Friedrich Gauss
[۳] Karatsuba

**مسأله ۵** مسئله‌ی ضرب ماتریس‌ها. دو ماتریس $X, Y$ با ابعاد $n \times n$ داریم که می‌خواهیم حاصل‌ضرب آنها را بدست آوریم. الگوریتمی بهینه برای اینکار پیشنهاد دهید.

**جواب ۵** الگوریتم بدیهی. در صورتیکه به ازای هر سطر از ماتریس اول و هر ستون از ماتریس دوم درایه‌های متناظر را در هم ضرب کنیم و نتیجه را ذخیره کنیم، زمان اجرای الگوریتم $O(n^۳)$ خواهد بود.

الگوریتم بهتر: برای حل این مسئله به روش تقسیم و حل، هر ماتریس را به ۴ ماتریس با اندازه‌ی $\frac{n}{۲} \times \frac{n}{۲}$ تقسیم می‌کنیم. بدین ترتیب هر زیرماتریس اولی باید در دو زیرماتریس دومی ضرب شود و نهایتا جواب‌ها با یکدیگر جمع شوند. درصورتیکه روابط بازگشتی را برای این تعداد حاصل‌ضرب بنویسیم الگوریتم پیشنهادی بهینه نخواهد بود و همانند مسئله‌ی قبل به جوابی مشابه الگوریتم بدیهی می‌رسیم. در این مسئله نیز روشی ارائه می‌دهیم تا تعداد حاصل‌ضرب‌ها را کاهش داده و از هشت به هفت حاصل‌ضرب برسانیم. این روش را استراسون [۴] مطرح کرده است.
ابتدا همانند زیر هر ماتریس را تقسیم‌بندی می‌کنیم:

$$X = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]$$

$$Y = \left[\begin{array}{c|c} E & F \\ \hline G & H \end{array}\right]$$

حال به روش زیر محاسبات را انجام می‌دهیم:

$$\begin{aligned}
P_۱ &= A \times (F - H) \\
P_۲ &= (A + B) \times H \\
P_۳ &= (C + D) \times E \\
P_۴ &= D \times (G - E) \\
P_۵ &= (A + D) \times (E + H) \\
P_۶ &= (B - D) \times (G + H) \\
P_۷ &= (A - C) \times (E + F)
\end{aligned}$$

و در نهایت برای رسیدن به حاصل‌ضرب نهایی مقادیر زیر را جایگذاری می‌کنیم:

$$X.Y = \left[\begin{array}{c|c} P_۵ + P_۴ - P_۲ + P_۶ & P_۱ + P_۲ \\ \hline P_۳ + P_۴ & P_۱ + P_۵ - P_۳ - P_۷ \end{array}\right]$$

که زمان اجرای برنامه نهایتا برابر است با: $T(n) = O(n^{\log_۲^۷}) = O(n^{۲/۸})$

**نکته ۱** می‌توان نشان داد با افراز به زیرماتریس‌های با اندازه نصف نمی‌توان تعداد ضرب‌های لازم را به کمتر از ۷ کاهش داد. با استفاده از زیرماتریس‌های با اندازه مناسب‌تر می‌توان به الگوریتم‌های بهتری برای محاسبه حاصل‌ضرب ماتریس‌ها دست یافت اما پیچیدگی هیچ‌یک از الگوریتم موجود $O(n^۲)$ نیست. پیش‌بینی می‌شود که می‌توان الگوریتم‌های با $O(n^{۲+\epsilon})$ با $\epsilon > ۰$ پیدا کرد و $\epsilon$ را کوچک و کوچک‌تر کرد.

[۴] Volker Strassen