به نام خدا

طراحی الگوریتم‌ها

آرش شفیعی

تحلیل الگوریتم‌ها

تحلیل الگوریتم‌ها

- آنالیز الگوریتم یا تحلیل الگوریتم [1] به معنای پیش بینی منابع مورد نیاز برای اجرای یک الگوریتم است. منابع مورد نیاز شامل زمان محاسبات، میزان حافظه، پهنای باند ارتباطی و مصرف انرژی می‌شود.
- معمولا برای یک مسئله الگوریتم‌های متعددی وجود دارند که هر یک می‌تواند از لحاظ تعدادی از معیارهای ارزیابی بهینه باشد.
- برای تحلیل الگوریتم از یک مدل محاسباتی استفاده می‌کنیم. در اینجا از مدل محاسباتی ماشین دسترسی تصادفی [2] استفاده می‌کنیم. در این مدل محاسباتی فرض می‌کنیم زمان مورد نیاز برای اجرای دستورات و دسترسی به حافظه، ثابت و به میزانی معین است.
- دستورات معمول در این مدل محاسباتی شامل دستورات محاسباتی ریاضی (مانند جمع و تفریق و ضرب و تقسیم و باقیمانده و کف و سقف)، دستورات جابجایی داده (مانند ذخیره، بارگیری و کپی) و دستورات کنترلی (مانند شرطی و انشعابی و فراخوانی تابع) می‌شوند.

---

[1] algorithm analysis
[2] random-access machine (RAM)

تحلیل الگوریتم‌ها

- عملیات محاسبه توان جزء دستورات اصلی مدل محاسباتی رم به حساب نمی‌آید، اما بسیاری از ماشین‌ها با عملیات انتقال بیت‌ها در زمان ثابت می‌توانند اعداد توانی را محاسبه کنند.
- همچنین در این مدل، سلسله مراتب حافظه مانند حافظه نهان [1] که در کامپیوترهای واقعی پیاده سازی شده است، وجود ندارد.
- مدل محاسباتی ماشین دسترسی تصادفی یک مدل ساده همانند ماشین تورینگ است که در آن دسترسی تصادفی به حافظه وجود دارد و عملیات ساده تعریف شده‌اند.

[1] cache memory

تحلیل الگوریتم‌ها

- تحلیل الگوریتم‌ها به منظور محاسبهٔ زمان اجرا و میزان حافظه مورد نیاز الگوریتم‌ها به کار می‌رود.
- زمان اجرا و میزان حافظهٔ مورد نیاز یک الگوریتم به ازای ورودی‌های مختلف متفاوت است و این مقادیر بر اساس اندازهٔ ورودی الگوریتم محاسبه می‌شوند.
- زمان اجرا و میزان حافظه مورد نیاز حافظه معیارهایی برای سنجش کارایی الگوریتم‌ها هستند.
- در این قسمت در مورد روش‌های مختلف تحلیل الگوریتم صحبت خواهیم کرد.
- عوامل زیادی در زمان اجرای یک الگوریتم تأثیر می‌گذارند که از آن جمله می‌توان به سرعت پردازنده، کامپایلر استفاده شده برای پیاده سازی الگوریتم، اندازهٔ ورودی الگوریتم و همچنین ساختار الگوریتم اشاره کرد.

تحلیل الگوریتم‌ها

- برخی ازین عوامل در کنترل برنامه نویس نیستند. برای مثال سرعت پردازنده عاملی است تأثیر گذار در سرعت اجرا که با پیشرفت صنعت سخت افزار بهبود می‌یابد و در کنترل برنامه نویس نیست. اما ساختار الگوریتم عاملی است که توسط طراح الگوریتم کنترل می‌شود و نقش مهمی در سرعت اجرا دارد.
- صرف نظر از عوامل فیزیکی، می‌توان سرعت اجرای برنامه را تابعی از اندازهٔ ورودی الگوریتم تعریف کرد که تعداد گام‌های لازم برای محاسبهٔ خروجی را بر اساس اندازه ورودی الگوریتم بیان می‌کند.
- تعداد گام‌های یک الگوریتم برای محاسبه یک مسئله به ساختار آن الگوریتم بستگی دارد.
- البته غیر از اندازهٔ ورودی، ساختار ورودی هم بر سرعت اجرای برنامه تأثیر گذار است. بنابراین سرعت اجرای برنامه را معمولاً در بهترین حالت (یعنی حالتی که ساختار ورودی به گونه‌ای است که الگوریتم کمترین زمان را برای اجرا بر روی یک ورودی با اندازه معین نیاز دارد) و بدترین حالت محاسبه می‌کنیم. همچنین می‌توان زمان اجرای برنامه را در حالت میانگین به دست آورد.

تحلیل الگوریتم‌ها

- برای تحلیل زمان مورد نیاز برای اجرای الگوریتم مرتب‌سازی، یک روش اجرای آن الگوریتم بر روی یک کامپیوتر و اندازه‌گیری زمان اجرا آن است.
- اما این اندازه‌گیری به ماشین مورد استفاده و کامپایلر و زبان برنامه نویسی مورد استفاده و اجرای برنامه‌های دیگر برروی آن ماشین بستگی دارد. نوع پیاده سازی و اندازهٔ ورودی نیز دو عامل دیگر در سرعت اجرای برنامهٔ مرتب سازی است.
- روش دیگر برای محاسبه زمان اجرای الگوریتم مرتب‌سازی، تحلیل خود الگوریتم است. در این روش محاسبه می‌کنیم هر دستور در برنامه چندبار اجرا می‌شوند. سپس فرمولی به دست آوریم که نشان دهندهٔ زمان اجرای برنامه است. این فرمول به اندازهٔ ورودی الگوریتم بستگی پیدا می‌کند ولی عوامل محیطی مانند سرعت پردازنده در آن نادیده گرفته می‌شود. از این روش می‌توان برای مقایسهٔ الگوریتم‌ها استفاده کرد.

تحلیل الگوریتم‌ها

- اندازهٔ ورودی [1] در بسیاری از مسائل مانند مسئله مرتب‌سازی تعداد عناصر تشکیل دهندهٔ ورودی است. در مسئله مرتب‌سازی اندازه ورودی در واقع تعداد عناصر آرایهٔ ورودی برای مرتب‌سازی است.
- در برخی از مسائل اندازهٔ ورودی در واقع تعداد بیت عدد صحیح ورودی است. برای مثال اندازهٔ ورودی مسئله تجزیهٔ یک عدد به عوامل اول، خود عدد ورودی است.
- در برخی مسائل تعداد ورودی‌ها بیش از یک پارامتر است، بنابراین اندازهٔ ورودی به بیش از یک پارامتر بستگی پیدا می‌کند. برای مثال در الگوریتم پیدا کردن کوتاه‌ترین مسیر در یک گراف، اندازهٔ ورودی تعداد رئوس و تعداد یال‌ها است.

[1] input size

تحلیل الگوریتم‌ها

- زمان اجرای [1] یک الگوریتم وابسته به تعداد دستورات اجرا شده و تعداد دسترسی‌ها به حافظه است. در هنگام محاسبات برای تحلیل الگوریتم فرض می‌کنیم برای اجرای یک دستور در برنامه به یک زمان ثابت نیاز داریم. یک دستور در اجراهای متفاوت ممکن است زمان اجرای متفاوتی داشته باشد ولی فرض می‌کنیم خط k ام برنامه، در زمان c_k اجرا شود.

- کل زمان اجرای یک برنامه، مجموع زمان اجرای همهٔ دستورات آن است. دستوری که m بار در کل برنامه تکرار می‌شود و در زمان c_k اجرا می‌شود، در کل به mc_k واحد زمان برای اجرا نیاز دارد.

- معمولاً زمان اجرای یک الگوریتم با ورودی n را با $T(n)$ نشان می‌دهیم.

[1] execution time

تحلیل الگوریتم مرتب‌سازی درجی

- الگوریتم مرتب‌سازی درجی را یک بار دیگر در نظر می‌گیریم.

Algorithm Insertion Sort

```
function INSERTION-SORT(A, n)
▷ A is an array of n elements
for i = 2 to n do
    key = A[i]
    j = i - 1
    while  j > 0 and A[j] > key  do
        A[j+1] = A[j]
        j = j-1
    A[j+1] = key
```

تحلیل الگوریتم مرتب‌سازی درجی

- برای محاسبه زمان اجرای الگوریتم مرتب‌سازی درجی، ابتدا تعداد تکرار هر یک از خط‌های برنامه را می‌شماریم.
- در این برنامه خط ۱ تعداد n بار و خطوط ۲ و ۳ و ۷ هر یک $n - 1$ بار تکرار می‌شوند.
- تعداد تکرار خطوط ۴ و ۵ و ۶ به تعداد تکرار حلقه بستگی دارد.
- زمان اجرای یک الگوریتم علاوه بر اندازه ورودی به ساختار ورودی نیز بستگی دارد. در الگوریتم مرتب‌سازی مسلماً مرتب‌سازی یک آرایهٔ مرتب شده از مرتب‌سازی یک آرایهٔ مرتب نشده سریع‌تر انجام می‌شود.

تحلیل الگوریتم مرتب‌سازی درجی

- زمان اجرای یک الگوریتم را معمولا در بهترین حالت و بدترین حالت محاسبه می‌کنیم. در بهترین حالت آرایهٔ ورودی الگوریتم مرتب شده است. بنابراین در بهترین حالت در هر بار اجرای خط ۴ ، برنامه از حلقه `while` خارج می‌شود و بنابراین خط ۴ تعداد $n-1$ بار اجرا می‌شود و خطوط ۵ و ۶ اجرا نمی‌شوند.

- زمان کل اجرای برنامه را می‌توانیم به صورت زیر بنویسیم.

$$
\begin{aligned}
T(n) &= c_1 + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\
&= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)
\end{aligned}
$$

- زمان اجرای این الگوریتم در بهترین حالت را می‌توانیم به صورت $an + b$ بنویسیم به ازای اعداد ثابت a و b و اندازه ورودی n بنابراین زمان اجرا در این حالت یک تابع خطی [1] از n است.

[1] linear function

تحلیل الگوریتم مرتب‌سازی درجی

- حال زمان اجرای الگوریتم مرتب سازی درجی را در بدترین حالت محاسبه می‌کنیم. در بدترین حالت آرایهٔ ورودی به صورت معکوس مرتب شده است و بنابراین هر یک از عناصر آرایه نیاز به بیشترین تعداد جابجایی دارد.
- در حلقهٔ `while` هر یک از عناصر $A[i]$ باید با همهٔ عناصر $A[1 : i-1]$ مقایسه شود بنابراین حلقه باید تعداد i بار به ازای $2, 3, ..., n$ تکرار شود.
- پس به طور کل خط ۴ باید به تعداد زیر تکرار شود.

$$\sum_{i=2}^{n} i = \Big(\sum_{i=1}^{n} i \Big) - 1 = \frac{n(n+1)}{2} - 1$$

تحلیل الگوریتم مرتب‌سازی درجی

- هر یک از خطوط ۵ و ۶ الگوریتم به ازای $i = 2, 3, ..., n$ تعداد $i - 1$ بار تکرار می‌شود.
- بنابراین برای خطوط ۵ و ۶ تعداد تکرار برابر است با :

$$\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

تحلیل الگوریتم مرتب سازی درجی

- زمان اجرای برنامه در بدترین حالت را می‌توانیم به صورت زیر محاسبه کنیم.

$$
\begin{aligned}
T(n) =& \; c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) \\
&+ \; c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1) \\
=& \left(\frac{c_4+c_5+c_6}{2}\right)n^2 + \left(c_1+c_2+c_3+\frac{c_4-c_5-c_6}{2}+c_7\right)n \\
&- \; (c_2+c_3+c_4+c_7)
\end{aligned}
$$

تحلیل الگوریتم مرتب‌سازی درجی

- بنابراین زمان اجرای الگوریتم مرتب سازی درجی در بدترین حالت را می‌توانیم به صورت $an^2 + bn + c$ بنویسیم به طوری که a و b و c اعداد ثابت و n ورودی برنامه است. پس زمان اجرای الگوریتم در بدترین حالت یک تابع مربعی [1] یا تابع درجه دوم از n است.

[1] quadratic function

تحلیل الگوریتم مرتب‌سازی درجی

- در حالت کلی از آنجایی که تعداد تکرارها در حلقه `while` مشخص نیست، زمان اجرای الگوریتم را می‌توانیم به صورت زیر بنویسیم که در آن t_i تعداد متغیر تکرارهای حلقه `while` است.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

تحلیل الگوریتم مرتب‌سازی درجی

- معمولاً در تحلیل الگوریتم‌ها، بدترین حالت [1] زمان اجرا را محاسبه می‌کنیم.

- دلیل این امر آن است که زمان اجرا در بدترین حالت در واقع یک کران بالا [2] برای زمان اجرا است و الگوریتم نمی‌تواند به زمانی بیشتر از آن نیاز داشته باشد. پس می‌توانیم تضمین کنیم که الگوریتم در زمانی که در بدترین حالت محاسبه کرده‌ایم اجرا می‌شود. همچنین در بسیاری از مواقع برای بسیاری از الگوریتم‌ها بدترین حالت بسیار اتفاق می‌افتد.

- دلیل دیگر برای تحلیل الگوریتم در بدترین حالت این است که زمان اجرا در بدترین حالت و در حالت میانگین [3] تقریبا معادل یکدیگرند. برای مثال در الگوریتم مرتب‌سازی درجی، در حالت میانگین در حلقهٔ `while` هر یک از $A[i]$ ها باید با نیمی از عناصر $A[1 : i-1]$ مقایسه شوند. بنابراین $t_i = i/2$. اگر کل زمان اجرا در حالت میانگین را محاسبه کنیم، زمان اجرا یک تابع درجه دوم از اندازهٔ ورودی به دست می‌آید. بنابراین زمان اجرا در بدترین حالت و حالت میانگین تقریبا برابرند.

---

[1] worst case

[2] upper bound

[3] average case

تحلیل مجانبی

- در تحلیل الگوریتم‌ها معمولاً در مورد مرتبهٔ رشد [1] یا نرخ رشد توابع [2] صحبت می‌کنیم و جزئیات را در محاسبات نادیده می‌گیریم. در واقع محاسبهٔ زمان اجرا را به صورت حدی در نظر می‌گیریم وقتی که اندازهٔ ورودی بسیار بزرگ باشد. وقتی n به بینهایت میل کند هر تابع درجه دوم با n^2 برابر است. در این حالت می‌گوییم زمان اجرا برنامه از مرتبه n^2 است.

- برای نشان دادن مرتبه بزرگی از حروف یونانی Θ (تتا) استفاده می‌کنیم. می‌گوییم زمان اجرای مرتب‌سازی درجی در بهترین حالت برابر است با $\Theta(n)$ و زمان اجرای آن در بدترین حالت برابر است با $\Theta(n^2)$، بدین معنی که برای n های بسیار بزرگ زمان اجرای الگوریتم در بدترین حالت تقریبا برابر است با n^2.

- زمان اجرای یک الگوریتم از یک الگوریتم دیگر بهتر است اگر زمان اجرای آن در بدترین حالت مرتبه رشد کمتری [3] داشته باشد.

---

[1] order of growth
[2] rate of growth
[3] lower order of growth

تحلیل مجانبی

- مرتبه رشد [1] زمان اجرای یک الگوریتم، معیار مناسبی برای سنجش کارایی [2] یک الگوریتم است که به ما کمک می‌کند یک الگوریتم را با الگوریتم‌های جایگزین آن مقایسه کنیم.
- گرچه محاسبه دقیق زمان اجرا در بسیاری مواقع ممکن است، اما این دقت در بسیاری مواقع ارزش افزوده‌ای ندارد چرا که به ازای ورودی‌های بزرگ مرتبه رشد زمان اجرا تعیین کننده مقدار تقریبی زمان اجرا است.
- تحلیل مجانبی [3] در آنالیز ریاضی روشی است برای توصیف رفتار حدی توابع. در تحلیل الگوریتم‌ها نیز می‌خواهیم تابع زمان اجرا را با استفاده از تحلیل مجانبی بررسی کنیم تا زمان اجرا را وقتی ورودی الگوریتم بدون محدودیت بزرگ می‌شود بسنجیم.

[1] order of growth
[2] efficiency
[3] asymptotic analysis

تحلیل مجانبی

- نماد O [1] در تحلیل مجانبی توابع، کران بالای [2] یک تابع را مشخص می‌کند. به عبارت دیگر با استفاده از این نماد می‌گوییم یک تابع از تابعی که با نماد O مشخص شده است سریع‌تر رشد نمی‌کند.
- برای مثال می‌گوییم تابع $2n^3 + 3n^2 + n + 4$ دارای کران بالای n^3 است و می‌نویسیم این تابع $O(n^3)$ است.
- همچنین می‌توانیم بگوییم این تابع $O(n^4)$ و $O(n^5)$ و به طور کلی $O(n^c)$ به ازای $c \geqslant 3$ است، چرا که سرعت رشد آن از این تابع بیشتر نیست.

[1] O-notation
[2] upper bound

تحلیل مجانبی

- نماد O کران بالای مجانبی [1] یک تابع را مشخص می‌کند. از نماد O برای تعیین کران بالای یک تابع استفاده می‌کنیم.
- به ازای تابع دلخواه $g(n)$ ، مجموعهٔ $O(g(n))$ شامل همهٔ توابعی است که کران بالای آنها $g(n)$ است و به صورت زیر تعریف می‌شود.

$$O(g(n)) = \{f(n) : \exists c, n_0 > 0, \forall n \geqslant n_0, 0 \leqslant f(n) \leqslant cg(n)\}$$

[1] asymptotic upper bound

تحلیل مجانبی

- به عبارت دیگر تابع $f(n)$ به مجموعه توابع $O(g(n))$ تعلق دارد اگر عدد مثبت c وجود داشته باشد به طوری‌که به ازای اعداد n بزرگ‌تر از n_0 داشته باشیم $f(n) \leqslant cg(n)$.
- طبق این تعریف توابع $f(n)$ باید توابع غیر منفی باشند.
- از آنجایی که نماد O در واقع یک مجموعه را تعریف می‌کند می‌توانیم بنویسیم $f(n) \in O(g(n))$ ، اما گاهی برای سادگی می‌نویسیم $f(n) = O(g(n))$ و می‌خوانیم $f(n)$ از $O(g(n))$ است، یا $g(n)$ کران بالای تابع $f(n)$ است.
- برای مثال $4n^2 + 100n + 500 = O(n^2)$. باید نشان دهیم c و n_0 وجود دارند که در شرایط تعریف شده صدق می‌کنند. به عبارت دیگر $4n^2 + 100n + 500 \leqslant cn^2$ به ازای $n_0 = 1$ برای اینکه این نامعادله درست باشد داریم $c = 604$.

تحلیل مجانبی

- نماد Ω [1] یا نماد اومگا کران پایین [2] یک تابع را در تحلیل مجانبی مشخص می‌کند. به عبارت دیگر با استفاده از این نماد می‌گوییم یک تابع از تابعی که با نماد Ω تعیین شده سریع‌تر رشد می‌کند.
- برای مثال می‌گوییم تابع $2n^3 + 3n^2 + n + 4$ دارای کران پایین n^3 است و می‌نویسیم این تابع $\Omega(n^3)$ است.
- همچنین می‌توانیم بگوییم این تابع $\Omega(n^2)$ و $\Omega(n)$ و به طورکلی $\Omega(n^c)$ به ازای $c \geqslant 3$ است.

---

[1] Ω-notation
[2] lower bound

تحلیل مجانبی

- نماد Ω کران پایین مجانبی [1] یک تابع را مشخص می‌کند. از نماد Ω برای تعیین کران پایین یک تابع استفاده می‌کنیم.

- به ازای یک تابع دلخواه $g(n)$ ، مجموعهٔ $\Omega(g(n))$ شامل همهٔ توابعی است که کران پایین آنها $g(n)$ است و به صورت زیر تعریف می‌شود.

$$\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0, \forall n \geqslant n_0, 0 \leqslant cg(n) \leqslant f(n)\}$$

- برای مثال $4n^2 + 100n + 500 = \Omega(n^2)$. به عبارت دیگر $4n^2 + 100n + 500 \geqslant cn^2$ به ازای همهٔ n_0 های مثبت این نامعادله درست است اگر $c = 4$

[1] asymptotic lower bound

تحلیل مجانبی

- نماد Θ [1] یا نماد تتا، کران اکید [2] یک تابع در تحلیل مجانبی را مشخص می‌کند. به عبارت دیگر با استفاده از این نماد می‌گوییم یک تابع دقیقا با یک نرخ تعیین شده رشد می‌کند، نه سریع‌تر و نه کندتر.
- اگر نشان دهیم یک تابع دارای کران بالای $f(n)$ و دارای کران پایین $f(n)$ است و یا عبارت دیگر $O(f(n))$ و $\Omega((f(n)))$ است، آنگاه آن تابع دقیقا از مرتبه $f(n)$ است و یا به عبارت دیگر $\Theta(f(n))$ است.
- برای مثال می‌گوییم تابع $2n^3 + 3n^2 + n + 4$ از مرتبه n^3 است و می‌نویسیم این تابع $\Theta(n^3)$ است.

[1] Θ-notation
[2] tight bound

تحلیل مجانبی

- نماد Θ کران اکید مجانبی [1] را مشخص می‌کند.
- به ازای تابع دلخواه $g(n)$ ، مجموعه $\Theta(g(n))$ شامل همهٔ توابعی است که کران اکید آنها $g(n)$ است، یعنی همه توابعی که $g(n)$ هم کران بالای آنها است و هم کران پایین آنها.
- به عبارت دیگر

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0, \forall n \geqslant n_0, 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\}$$

- می‌توانیم ثابت کنیم که به ازای دو تابع $f(n)$ و $g(n)$ داریم $f(n) \in \Theta(g(n))$ اگر و تنها اگر $f(n) \in O(g(n))$ و $f(n) \in \Omega(g(n))$.

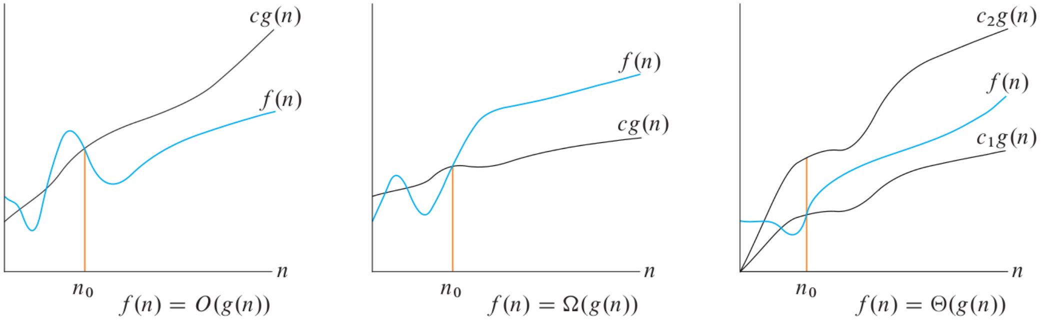
[1] asymtotically tight bound

تحلیل مجانبی

- نمادهای O، Ω، و Θ بر روی توابع گسسته عمل می‌کنند، یعنی توابعی که دامنهٔ آنها بر روی اعداد صحیح $\mathbb{N}$ و برد آنها بر روی اعداد حقیقی $\mathbb{R}$ تعریف شده است. از این نمادها برای تحلیل مجانبی زمان اجرای الگوریتم‌ها یعنی $T(n)$ استفاده می‌کنیم.

تحلیل مجانبی

- در شکل زیر مفاهیم نمادهای مجانبی نشان داده شده‌اند.



تحلیل مجانبی الگوریتم مرتب‌سازی درجی

- حال الگوریتم مرتب‌سازی درجی را یک بار دیگر در نظر می‌گیریم.

Algorithm Insertion Sort

```
function INSERTION-SORT(A, n)
▷ A is an array of n elements
for i = 2 to n do
   key = A[i]
   j = i - 1
   while  j > 0 and A[j] > key  do
      A[j+1] = A[j]
      j = j-1
   A[j+1] = key
```

تحلیل مجانبی الگوریتم مرتب‌سازی درجی

- این الگوریتم در یک حلقه `for` به تعداد $n-1$ بار تکرار می‌شود. به ازای هر بار تکرار در این حلقه یک حلقه درونی `while` وجود دارد که در بدترین حالت $i-1$ بار تکرار می‌شود و i حداکثر n است بنابراین تعداد کل تکرارها حداکثر $(n-1)(n-1)$ است، که این مقدار از n^2 کوچکتر است. بنابراین می‌توانیم بگوییم زمان اجرای این الگوریتم $O(n^2)$ است.

تحلیل مجانبی الگوریتم مرتب‌سازی درجی

- حال می‌خواهیم نشان دهیم زمان اجرای این الگوریتم در بدترین حالت $\Omega(n^2)$ است. برای این کار باید نشان دهیم حداقل یک ورودی وجود دارد که زمان اجرای آن حداقل از مرتبه n^2 است.

- فرض کنید یکی از ورودی‌های الگوریتم، آرایه‌ای است که طول آن مضرب ۳ است و در این ورودی بزرگ‌ترین عناصر آرایه در یک سوم ابتدای آرایه قرار دارند. برای این‌که این آرایه مرتب شود همهٔ این عناصر باید به یک‌سوم انتهای آرایه انتقال پیدا کنند. برای این انتقال حداقل هر عنصر باید $n/3$ بار به سمت راست حرکت کند تا از ثلث میانی آرایه عبور کند. این انتقال باید برای حداقل یک‌سوم عناصر اتفاق بیافتد، پس زمان اجرا در این حالت حداقل $(n/3)(n/3)$ است یا به عبارت دیگر $\Omega(n^2)$ است.

- از آنجایی که مرتبه رشد مرتب‌سازی درجی در بدترین حالت حداکثر و حداقل از مرتبه n^2 است یعنی مرتبه رشد آن $O(n^2)$ و $\Omega(n^2)$ است، بنابراین می‌توانیم نتیجه بگیریم مرتبه رشد آن در بدترین حالت از مرتبه n^2 است یا به عبارت دیگر $\Theta(n^2)$ است.

تحلیل سرشکنی

- فرض کنید عضو یک باشگاه می‌شوید. باشگاه از شما مبلغی به ازای حق عضویت دریافت می‌کند که باید ماهیانه بپردازید. اما در هر بار استفاده از باشگاه نیز باید مبلغی پرداخت کنید. فرض کنید حق عضویت ۶۰ تومان است و در هربار استفاده باید ۳ تومان بپردازد. اگر بخواهید هر روز از باشگاه استفاده کنید در واقع باید ماهیانه ۱۵۰ تومان یا به عبارت دیگر روزی ۵ تومان بپردازید.
- وقتی هزینه را به طور میانگین به ازای واحدهای کوچک‌تر محاسبه می‌کنیم می‌گوییم هزینه‌ها را سرشکن [1] می‌کنیم.

[1] amortize

تحلیل سرشکنی

- همچنین هنگامی که زمان اجرای یک الگوریتم را محاسبه می‌کنیم، می‌توانیم میانگین لازم را برای انجام عملیات محاسبه کنیم. چنین تحلیلی، تحلیل سرشکن [1] گفته می‌شود. در تحلیل سرشکن الگوریتم، زمان کل اجرا بر تعداد عملیات تقسیم می‌شود و زمان لازم برای اجرای یک عمل به دست می‌آید.
- تحلیل سرشکن، کارایی هر یک از عملیات را به طور متوسط مشخص می‌کند. به عبارت دیگر، اگر تعدادی از عملیات به زمان اجرای زیادی لازم داشته باشند و بقیه عملیات زمان زیادی را صرف نکنند، با تقسیم زمان اجرای کل بر تعداد عملیات نشان می‌دهیم به طور متوسط هریک از عملیات در چه زمانی اجرا می‌شوند.

[1] amortize analysis

تحلیل تجمعی

- در تحلیل تجمعی [1] ، نشان می‌دهیم دنباله‌ای از n عملیات در بدترین حالت به زمان $T(n)$ نیاز دارد.
- بنابراین در بدترین حالت، هزینهٔ متوسط یا هزینهٔ سرشکن، به ازای هریک از عملیات برابراست با $T(n)/n$.
- هزینهٔ به دست آمده، هزینهٔ متوسطی است که به ازای هر یک از عملیات نیاز است حتی اگر برخی از عملیات به زمان کمتری نیاز داشته باشند.

[1] aggregate analysis

تحلیل تجمعی

- می‌خواهیم عملیات مربوط به یک پشته را با استفاده از تحلیل تجمعی تحلیل کنیم.
- تابع $\mathrm{Push}(S, x)$ شیء x را در پشته S قرار می‌دهد و تابع $\mathrm{Pop}(S)$ یک شیء از پشته استخراج می‌کند. فراخوانی Pop با پشتهٔ خالی منجر به خطا می‌شود.

تحلیل تجمعی

- هزینهٔ هریک از عملیات پشته $O(1)$ است. فرض کنیم هزینهٔ انجام این عمل به میزان ۱ واحد زمان باشد. مجموع هزینه‌های دنباله‌ای از n عملیات Push و Pop برابراست با n و زمان اجرای n عملیات $\Theta(n)$ است.

- حال یک عملگر جدید به نام $\mathrm{Multipop}(S, k)$ می‌افزاییم که با استفاده از آن k شیء از روی پشته S برداشته می‌شود و در صورتی که تعداد اشیای درون پشته کمتر از k باشد، همهٔ اشیای پشته حذف می‌شوند.

تحلیل تجمعی

- الگوریتم تابع Multipop به صورت زیر است.

Algorithm Multipop

```
function MULTIPOP(S, k)
while not Stack-Empty(S) and k > 0 do
    Pop(S)
    k = k - 1
```

تحلیل تجمعی

- حال می‌خواهیم زمان اجرای $\mathrm{Multipop}(S, k)$ را بر روی یک پشته با s شیء محاسبه کنیم.

- زمان اجرای این تابع وابسته به زمان اجرای Pop است. تعداد تکرارهای حلقه در این الگوریتم برابراست با $\min\{s, k\}$ و چون Pop به زمان ثابت برای اجرا نیاز دارد، تعداد تکرارهای کل برابر می‌شود با $\min\{s, k\}$.

تحلیل تجمعی

- حال دنباله‌ای از n عملیات Push و Pop و $\mathrm{Multipop}$ را در نظر بگیرید. فرض کنید یک پشتهٔ خالی داریم و می‌خواهیم این عملیات را بر روی پشته انجام دهیم.
- اگر بخواهیم زمان اجرای کل عملیات را تحلیل کنیم، می توانیم بگوییم در بدترین حالت n عملیات $\mathrm{Multipop}$ داریم که هرکدام $O(n)$ زمان می‌برند و بنابراین در مجموع زمان اجرا $O(n^2)$ است.
- اما اگر بخواهیم با تحلیل تجمعی زمان اجرا این عملیات را تحلیل کنیم، می‌گوییم عملیات $\mathrm{Multipop}$ بدون انجام عملیات Push نمی‌تواند انجام بگیرد، بنابراین در بدترین حالت در مجموع به $O(n)$ زمان نیاز داریم و اگر این زمان را به تعداد عملیات تقسیم کنیم، زمان میانگین به ازای هر یک از عملیات برابراست با $O(n)/n = O(1)$

تحلیل تجمعی

- یک مثال دیگر از تحلیل تجمعی را بررسی می‌کنیم. یک شمارندۀ دودویی k بیتی را در نظر بگیرید که از صفر شروع به شمارش می‌کند.
- فرض کنید برای شمارنده از آرایۀ $A[0 : k-1]$ استفاده کنیم. توسط آرایۀ A عدد x نشان داده می‌شود به طوری‌که

$$x = \sum_{i=0}^{k-1} A[i] \times 2^i$$

تحلیل تجمعی

- می‌خواهیم الگوریتمی بنویسیم که مقدار این شمارندهٔ k بیتی را یک واحد افزایش دهد. الگوریتم زیر نحوهٔ اجرای این شمارنده را نشان می‌دهد.

Algorithm Increment

```
function INCREMENT(A, k)
i = 0
while i < k and A[i] == 1 do
    A[i] = 0
    i = i + 1
if i < k then
    A[i] = 1
```

تحلیل تجمعی

- شکل زیر مقدار آرایهٔ A را به ازای هر یک از اعداد شمارنده نشان می‌دهد. هزینهٔ افزایش شمارنده (تعداد تکرارهای حلقه در الگوریتم شمارش) به ازای هر شمارش و همچنین طور تجمعی در سمت راست نشان داده شده است.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	cost	Total cost
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1	1
2	0	0	0	0	0	0	1	0	2	3
3	0	0	0	0	0	0	1	1	1	4
4	0	0	0	0	0	1	0	0	3	7
5	0	0	0	0	0	1	0	1	1	8
6	0	0	0	0	0	1	1	0	2	10
7	0	0	0	0	0	1	1	1	1	11
8	0	0	0	0	1	0	0	0	4	15
9	0	0	0	0	1	0	0	1	1	16
10	0	0	0	0	1	0	1	0	2	18
11	0	0	0	0	1	0	1	1	1	19
12	0	0	0	0	1	1	0	0	3	22
13	0	0	0	0	1	1	0	1	1	23
14	0	0	0	0	1	1	1	0	2	25
15	0	0	0	0	1	1	1	1	1	26
16	0	0	0	1	0	0	0	0	5	31

تحلیل تجمعی

- فرض کنید می‌خواهیم n واحد به شمارنده بیافزاییم.
- در اینجا نیز با یک تحلیل ساده می‌توانیم زمان اجرا را به دست آوریم.
- یک اجرای الگوریتم در بدترین حالت در زمان $\Theta(k)$ اجرا می‌شود، یعنی وقتی همهٔ بیت‌ها ۱ باشند.
- بنابراین دنباله‌ای از n عملیات به زمان $O(nk)$ در بدترین حالت نیاز دارد.
- اما اگر بخواهیم دقیق‌تر این الگوریتم را تحلیل کنیم، می‌بینیم $A[0]$ به ازای هر واحد افزایش یک بار تغییر می‌کند، $A[1]$ به ازای هر دو واحد افزایش شمارنده یک بار تغییر می‌کند، $A[2]$ به ازای هر چهار واحد افزایش شمارنده یک بار تغییر می‌کند، الی آخر.

تحلیل تجمعی

- بنابراین پس از n واحد افزایش شمارنده، $A[0]$ تعداد n بار، $A[1]$ تعداد $\lfloor n/2 \rfloor$ بار، و $A[2]$ تعداد $\lfloor n/4 \rfloor$ بار ، و به طور کل $A[i]$ تعداد $\lfloor n/2^i \rfloor$ تغییر می‌کند.

- بنابراین در مجموع برای n واحد افزایش شمارنده k بیتی، تعداد تغییرات بیت‌ها به صورت زیر محاسبه می‌شوند.

$$\sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

- بنابراین در مجموع این شمارنده در زمان $O(n)$ اجرا می‌شود و میانگین زمان اجرا و هزینه سرشکن به ازای یک عملیات برابراست با $O(n)/n = O(1)$.