به نام خدا

طراحی الگوریتم‌ها

آرش شفیعی

الگوریتم‌های تقسیم و حل

طراحی الگوریتم با استقرا

- استقرای ریاضی [1] روشی است برای اثبات درستی گزاره $P(n)$ برای همه اعداد طبیعی n . به عبارت دیگر هنگامی که می‌خواهیم درستی گزاره‌های $P(0)$ ، $P(1)$ ، $P(2)$ ، $\cdots$ را ثابت کنیم، می‌توانیم از استقرا استفاده کنیم.
- به زبان استعاری با استفاده از استقرا ثابت می‌کنیم که می‌توانیم هر نردبانی را با طول دلخواه یا بینهایت بالا برویم اگر ثابت کنیم که می‌توانیم برروی پله اول برویم (پایهٔ استقرا [2]) و همچنین ثابت کنیم اگر برروی پلهٔ n بودیم می‌توانیم برروی پله $n+1$ نیز گام بگذاریم (گام استقرا [3]).
- بنابراین در روش استقرایی برای اثبات درستی $P(n)$ باید ثابت کنیم $P(1)$ درست است (پایهٔ استقرا) و همچنین اگر $P(n)$ درست باشد، آنگاه $P(n+1)$ نیر درست است (گام استقرا).

[1] induction
[2] base case
[3] induction step

طراحی الگوریتم با استقرا

- در طراحی یک مسئله به روش استقرایی، باید برای پاسخ مسئله یک رابطه پیدا کنیم و پاسخ مسئله را به روش استقرایی اثبات کنیم. و سپس می‌توانیم از رابطهٔ به دست آمده برای حل مسئله استفاده کنیم.
- برای مثال فرض کنید می‌خواهیم جمع n عدد اول صحیح را به دست آوریم. برای این کار n عدد را با یکدیگر جمع کنیم. پس الگوریتم در واقع $O(n)$ است.
- برای حل این مسئله به روش استقرایی باید رابطه‌ای برای جواب مسئله پیدا کنیم. به عبارت دیگر آیا عبارتی وجود دارد که توسط آن بتوان جمع n عدد اول اعداد صحیح را به دست آورد؟

طراحی الگوریتم با استقرا

- با استفاده از استقرا می‌توان اثبات کرد.

$$P(n) = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

- به عبارت دیگر باید اثبات کنیم $P(1) = \frac{1(2)}{2}$ درست است و همچنین اگر $P(n) = \frac{n(n+1)}{2}$ باشد آنگاه $P(n+1) = \frac{(n+1)(n+2)}{2}$ نیز درست است.

طراحی الگوریتم با استقرا

- اثبات : می‌دانیم $P(n+1) = P(n) + (n+1)$ بنابراین $P(n+1) = \frac{n(n+1)}{2} + (n+1)$.
- با بسط این رابطه به دست می‌آوریم $P(n+1) = \frac{(n+1)(n+2)}{2}$.
- حال برای محاسبهٔ n عدد اول عددی صحیح کافی است از رابطه $P(n)$ استفاده کنیم. این الگوریتم در زمان $O(1)$ انجام می‌شود.

الگوریتم‌های تقسیم و حل

- برای حل یک مسئله به روش‌های متنوعی می‌توان الگوریتم طراحی کرد.
- الگوریتم مرتب‌سازی درجی یک الگوریتم ساده است که به روش افزایشی با مرتب‌سازی زیر آرایه‌های کوچک‌تر آرایه آغاز می‌شود و در نهایت کل آرایه را مرتب می‌کند. در واقع به ازای هر عنصر `A[i]` ، این عنصر در مکان مناسب خود در زیر آرایه مرتب شدهٔ `A[1 : i-1]` قرار می‌گیرد.

الگوریتم‌های تقسیم و حل

- در این قسمت با روشی دیگر برای حل مسئله‌های محاسباتی آشنا می‌شویم، که به آن روش تقسیم و حل [1] گفته می‌شود و الگوریتم‌هایی که از این روش استفاده می‌کنند، در دستهٔ الگوریتم‌های تقسیم و حل قرار می‌گیرند.
- از روش تقسیم و حل برای حل مسئلهٔ مرتب‌سازی استفاده می‌کنیم و زمان اجرای آن را محاسبه می‌کنیم.
- خواهیم دید که با استفاده از این روش، مسئلهٔ مرتب‌سازی در زمان کمتری نسبت به الگوریتم مرتب‌سازی درجی حل می‌شود.

[1] divide and conquer method

الگوریتم‌های تقسیم و حل

- بسیاری از الگوریتم‌های کامپیوتری بازگشتی [1] هستند. در یک الگوریتم بازگشتی، برای حل یک مسئله با یک ورودی معین ، خود الگوریتم با ورودی یا ورودی‌های کوچکتر فراخوانی می‌شود.
- برای مثال، برای به دست آوردن فاکتوریل عدد n کافی است فاکتوریل عدد n−۱ را فراخوانی کنیم.
- به الگوریتم‌هایی که ورودی مسئله را تقسیم می‌کنند و به طور بازگشتی الگوریتم را برای قسمت‌های تقسیم شده فراخوانی می‌کنند، الگوریتم‌های تقسیم و حل گفته می‌شود.

[1] recursive

الگوریتم‌های تقسیم و حل

- به عبارت دیگر یک الگوریتم تقسیم و حل یک مسئله را به چند زیر مسئله تقسیم می‌کند که مشابه مسئلهٔ اصلی هستند و الگوریتم را برای زیر مسئله‌ها فراخوانی می‌کند و سپس نتایج به دست آمده از زیر مسئله‌ها را با هم ترکیب می‌کند تا نتیجهٔ نهایی برای مسئلهٔ اصلی به دست آید.

- معمولاً پس از شکسته شدن یک مسئله به زیر مسئله‌ها، زیر مسئله‌هایی به دست می‌آیند که می‌توانند دوباره شکسته شوند و این روند تا جایی ادامه پیدا می‌کند که مسئله امکان شکسته شدن نداشته باشد. وقتی مسئله امکان شکسته شدن نداشته باشد، حالت پایه [1] به دست می‌آید که حل مسئله در حالت پایه به سادگی امکان پذیر است.

[1] base case

الگوریتم‌های تقسیم و حل

- یک الگوریتم تقسیم و حل از سه مرحلهٔ زیر تشکیل شده‌است.

۱. تقسیم [1] : مسئله به چند زیر مسئله که نمونه‌های کوچکتر مسئلهٔ اصلی هستند تقسیم می‌شود.
۲. حل یا غلبه [2] : زیر مسئله‌ها به صورت بازگشتی حل می‌شوند.
۳. ترکیب [3] : زیر مسئله‌های حل شده با یکدیگر ترکیب می‌شوند تا جواب مسئلهٔ اصلی به دست بیاید.

[1] divide
[2] conquer
[3] combine

مرتب‌سازی ادغامی

- الگوریتم مرتب‌سازی ادغامی [1] در دستهٔ الگوریتم‌های تقسیم و حل قرار می‌گیرد. با شروع از آرایهٔ `A[1:n]` ، در هر مرحله یکی از زیر آرایه‌های `A[p:r]` مرتب می‌شود و سپس این زیر آرایه‌ها با یکدیگر ادغام می‌شوند تا آرایهٔ اصلی مرتب شود. برای هر یک از زیر آرایه‌ها، الگوریتم مرتب‌سازی ادغامی فراخوانی می‌شود و به همین نحو، آن زیر آرایه‌ها تقسیم شده و به روش بازگشتی مرتب می‌شوند.

[1] merge sort

مرتب‌سازی ادغامی

- مراحل انجام مرتب‌سازی ادغامی به صورت زیر است :

۱. تقسیم : آرایهٔ `A[p:r]` به دو زیرآرایهٔ مساوی تقسیم می‌شود. اگر q وسط p و r باشد، آنگاه دو آرایهٔ به دست آمده عبارتند از `A[p:q]` و `A[q+1:r]` . در مرحلهٔ اول p برابر با ۱ و r برابر است با n .

۲. حل : الگوریتم به صورت بازگشتی برای دو زیر آرایهٔ `A[p:q]` و `A[q+1:r]` فراخوانی می‌شود.

۳. ترکیب : با ادغام دو آرایهٔ `A[p:q]` و `A[q+1:r]` که هر دو مرتب شده هستند، آرایهٔ مرتب شدهٔ `A[p:r]` به دست می‌آید.

مرتب‌سازی ادغامی

- این الگوریتم به طور بازگشتی فراخوانی می‌شود تا به حالت پایه برسیم. در حالت پایه، آرایهٔ به دست آمده شامل تنها یک عنصر است که در این حالت آرایه نیاز به مرتب‌سازی ندارد. در واقع هنگامی به حالت پایه می‌رسیم که p برابر با r باشد.
- در مرحلهٔ ادغام، با فرض اینکه دو آرایهٔ به دست آمده مرتب شده هستند، دو آرایه باید به نحوی با یکدیگر ترکیب شوند که آرایهٔ به دست آمده مرتب شده باشد.

مرتب‌سازی ادغامی

- الگوریتم مرتب‌سازی ادغامی به صورت زیر است.

Algorithm Merge Sort

```
   function MERGE-SORT(A, p, r)
1: if p >= r then  ▷ zero or one element?
2:    return
3: q = ⌊ (p+r)/2 ⌋  ▷  midpoint of A[p:r]
4: Merge-Sort (A, p, q)  ▷  recursively sort A[p:q]
5: Merge-Sort (A, q+1, r)  ▷  recursively sort A[q+1:r]
6: Merge (A, p, q, r)  ▷ Merge A[p:q] and A[q+1:r] into A[p:r].
```

مرتب‌سازی ادغامی

- برای ادغام دو زیرآرایه از الگوریتم زیر استفاده می‌کنیم.

Algorithm Merge Sort

```
   function MERGE(A, p, q, r)
1: nl = q - p + 1  ▷ length of A[p:q]
2: nr = r - q  ▷ length of A[q+1 : r]
3: let L[ 0 : nl - 1 ] and R[ 0 : nr - 1 ] be new arrays
4: for i = 0 to nl - 1 do  ▷ copy A[p:q] into L[0:nl - 1]
5:    L[i] = A[p+i]
6: for j = 0 to nr - 1 do  ▷ copy A[q+1:r] into L[0:nr - 1]
7:    R[j] = A[q + j + 1]
```

مرتب‌سازی ادغامی

Algorithm Merge Sort

```
   function MERGE(A, p, q, r)
 8: i = 0  ▷ i indexes the smallest remaining element in L
 9: j = 0  ▷ j indexes the smallest remaining element in R
10: k = p  ▷ k indexes the location in A to fill
    ▷ As long as each of the arrays L and R contains un unmerged element,
   copy the smallest unmerged element back into A[p : r].
11: while  i < nl and j < nr do
12:    if L[i] <= R[j] then
13:       A[k] = L[i]
14:       i = i + 1
15:    else
16:       A[k] = R[j]
17:       j = j + 1
18:    k = k + 1
```

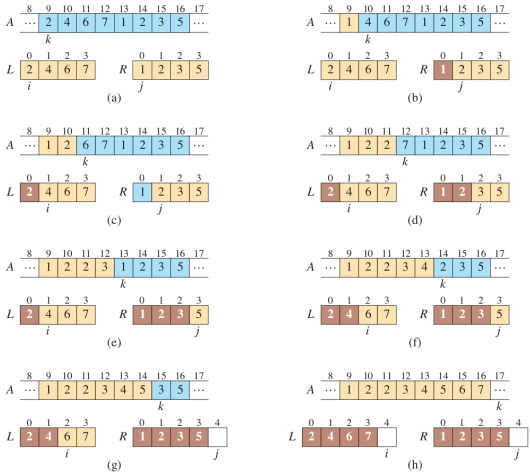
مرتب‌سازی ادغامی

Algorithm Merge Sort

```
   function MERGE(A, p, q, r)
    ▷ Having gone through one of L and R entirely, copy the remainder of
   the other to the end of A[p:r]
18: while i < nl do
19:    A[k] = L[i]
20:    i = i + 1
21:    k = k + 1
22: while  j < nr do
23:    A[k] = R[j]
24:    j = j + 1
25:    k = k + 1
```

مرتب‌سازی ادغامی

- یک مثال از ادغام دو زیر آرایه در شکل زیر نشان داده شده‌است.

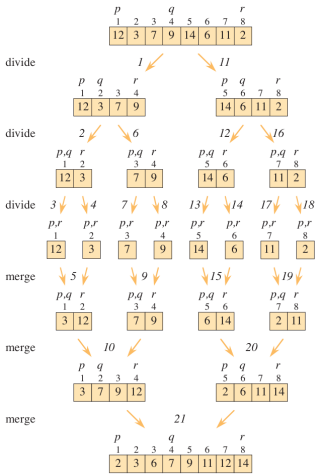


مرتب‌سازی ادغامی

- در حلقهٔ تکرار الگوریتم ادغام، در هر تکرار یکی از عناصر در آرایهٔ A کپی می‌شوند و درکل تا پایان الگوریتم n عنصر در آرایه کپی می‌شوند، پس زمان اجرای این الگوریتم $\Theta(n)$ است.

مرتب‌سازی ادغامی

- یک مثال مرتب‌سازی ادغامی در شکل زیر نشان داده شده‌است.



مرتب‌سازی ادغامی

- وقتی یک مسئله به صورت بازگشتی طراحی می‌شود، زمان اجرای آن را نیز معمولا با استفاده از معادلات بازگشتی [1] به دست می‌آوریم. در این معادلات بازگشتی، زمان اجرای یک الگوریتم با ورودی اندازهٔ n توسط زمان اجرای همان الگوریتم با ورودی‌هایی از اندازه‌های کوچک‌تر به دست می‌آید. روش‌های متعددی برای حل مسائل بازگشتی وجود دارند که می‌توان از آنها استفاده کرد.

[1] recurrence equation

مرتب‌سازی ادغامی

- به طور کلی اگر فرض کنیم زمان اجرا یک الگوریتم برای ورودی با اندازهٔ n برابر با $T(n)$ باشد و توسط روش تقسیم و حل مسئلهٔ مورد نظر به a زیر مسئله تقسیم شود که اندازه ورودی هر کدام n/b باشد، آنگاه به زمان $aT(n/b)$ برای حل مسئله نیاز داریم.
- همچنین اگر به زمان $D(n)$ برای تقسیم مسئله به زیر مسئله‌ها و به زمان $C(n)$ برای ادغام زیر مسئله‌ها نیاز داشته باشیم، آنگاه این زمان‌ها به زمان مورد نیاز برای حل مسئله افزوده می‌شوند.

مرتب‌سازی ادغامی

- فرض کنیم در حالت پایه، یعنی وقتی اندازهٔ ورودی از یک مقدار معین کوچکتر است، اجرای برنامه در زمان ثابت انجام شود، یعنی زمان اجرای برنامه در حالت پایه به اندازهٔ ورودی n بستگی نداشته باشد.
- در حالت کلی زمان اجرای یک الگوریتم تقسیم و حل را می‌توانیم با استفاده از رابطهٔ بازگشتی زیر بنویسیم.

$$T(n) = \begin{cases} \Theta(1) & \text{اگر } n < n_0 \\ D(n) + aT(n/b) + C(n) & \text{در باقی حالات} \end{cases}$$

مرتب‌سازی ادغامی

- حال زمان اجرای الگوریتم مرتب‌سازی ادغامی را در بدترین حالت به ازای یک آرایهٔ با طول n تحلیل می‌کنیم.
۱. تقسیم : تقسیم کردن آرایه به دو قسمت در زمان ثابت انجام می‌شود، بنابراین داریم $D(n) = \Theta(1)$
۲. حل : در مرحلهٔ حل از دو آرایه با اندازه $n/2$ به صورت بازگشتی استفاده می‌کنیم بنابراین زمان مورد نیاز در این مرحله برابر است با $2T(n/2)$. توجه کنید که ممکن است آرایه بر دو بخش پذیر نباشد، اما معمولاً در تحلیل الگوریتم از توابع کف و سقف صرف نظر می‌کنیم، چرا که تأثیری در تحلیل الگوریتم نمی‌گذارند.
۳. ترکیب : در این مرحله برای ادغام دو آرایه، جهت تولید یک آرایه با طول n به زمان $\Theta(n)$ نیاز داریم، بنابراین داریم $C(n) = \Theta(n)$

مرتب‌سازی ادغامی

- بنابراین در مجموع زمان اجرای الگوریتم مرتب‌سازی ادغامی به صورت زیر است :

$$T(n) = 2T(n/2) + \Theta(n)$$

- با حل این معادله بازگشتی می‌توان به دست آورد $T(n) = \Theta(n \lg n)$ ، بنابراین زمان مورد نیاز برای اجرای الگوریتم مرتب‌سازی ادغامی از مرتب‌سازی درجی بهتر است.

مرتب‌سازی ادغامی

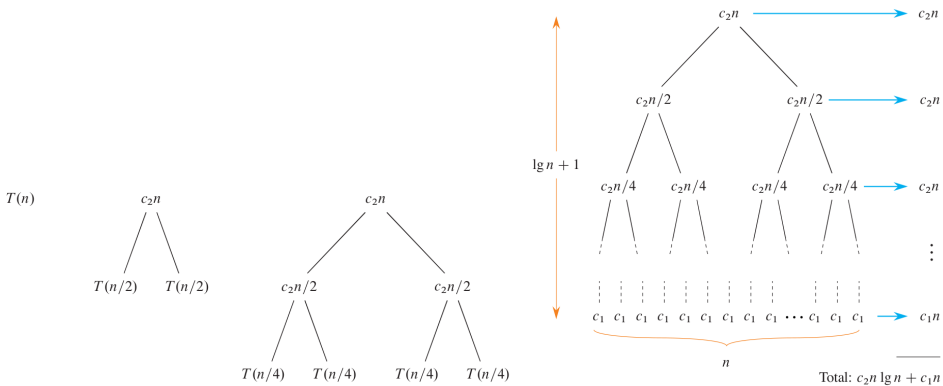
- حال برای اینکه بدون حل معادله بازگشتی، زمان اجرای به دست آمده را درک کنیم، می‌توانیم الگوریتم را به صورت زیر تحلیل کنیم.
- برای سادگی فرض می‌کنیم طول آرایهٔ ورودی برابر با n بوده و n توانی از ۲ است. با این ساده‌سازی همیشه با تقسیم n بر ۲ یک عدد صحیح به دست می‌آید.
- زمان اجرای الگوریتم را به صورت زیر می‌نویسیم.

$$T(n) = \begin{cases} c_1 & \text{اگر } n = 1 \\ 2T(n/2) + c_2 n & \text{اگر } n > 1 \end{cases}$$

- در اینجا c_1 زمان اجرای الگوریتم است هنگامی که طول ورودی ۱ باشد و c_2 مضرب ثابتی است که برای تقسیم و ادغام آرایه با طول n نیاز داریم.

مرتب‌سازی ادغامی

- شکل‌های زیر تقسیم این مسئله را به زیر مسئله‌ها و تحلیل زمان زیر مسئله‌ها را نشان می‌دهد.



مرتب‌سازی ادغامی

- مقدار c_2n در ریشهٔ این درخت در واقع زمان مورد نیاز برای تقسیم و ادغام را نشان می‌دهد، هنگامی که اندازهٔ مسئله برابر است با n . دو زیر درخت در سطح ۱ این درخت زمان‌های مورد نیاز را وقتی اندازهٔ ورودی $n/2$ است نشان می‌دهند. هزینه مورد نیاز برای تقسیم و ادغام هرکدام از این زیر درخت‌ها برابر است با $c_2n/2$ و مجموعه این هزینه‌ها برای دو زیر درخت برابر است با c_2n.
- چنانچه این محاسبات را ادامه دهیم، به این نتیجه می‌رسیم که هزینه تقسیم و ادغام برای هر یک از سطوح درخت برابر است با c_2n.

مرتب‌سازی ادغامی

- در مرحله آخر، یعنی وقتی به برگ‌های درخت می‌رسیم، حالت پایه را نشان می‌دهد که در این حالت زمان اجرای هر یک از زیر آرایه‌ها برابر است با c_1 و چون تعداد n زیر آرایه با طول ۱ داریم، زمان اجرا برای کل زیر آرایه‌ها برابر است با $c_1 n$.
- از آنجایی که این درخت در هر مرحله به دو بخش تقسیم می‌شود، تعداد سطوح درخت برابر است با $\lg n + 1$.
- بنابراین زمان کل اجرای الگوریتم برابر است با $c_2 n \lg n + c_1 n$.
- می‌توانیم با استفاده از تحلیل مجانبی بنویسیم $T(n) = \Theta(n \lg n)$.

ضرب ماتریس‌ها

- از روش تقسیم و حل می‌توانیم برای ضرب ماتریس‌های مربعی استفاده کنیم.
- فرض کنیم $A = (a_{ij})$ و $B = (b_{ij})$ دو ماتریس $n \times n$ باشند. ماتریس $C = A \cdot B$ نیز یک ماتریس $n \times n$ است که درایه‌های آن به صورت زیر محاسبه می‌شوند.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

ضرب ماتریس‌ها

- الگوریتم ضرب دو ماتریس در زیر نوشته شده‌است.

Algorithm Matrix

```
   function MATRIX-MULTIPLY(A, B, C, n)
1: for i do = 1 to n  ▷ compute entries in each of n rows
2:    for j do = 1 to n     ▷ compute n entries in row i
3:       for k do = 1 to n
4:          c[i,j] = c[i,j] + a[i,k] * b[k,j]          ▷ compute c[i,j]
```

ضرب ماتریس‌ها

- از آنجایی که خط ۴ باید n^3 بار تکرار شود، بنابراین زمان مورد نیاز برای اجرای این الگوریتم برابر است با $\Theta(n^3)$.

ضرب ماتریس‌ها

- حال می‌خواهیم ضرب دو ماتریس را توسط روش تقسیم و حل محاسبه کنیم.
- در مرحله تقسیم یک ماتریس $n \times n$ را به چهار ماتریس $n/2 \times n/2$ تقسیم می‌کنیم. برای سادگی فرض می‌کنیم n توانی از ۲ باشد و تقسیم کردن آن به ۲ در فرایند الگوریتم تقسیم و حل وجود داشته باشد.

ضرب ماتریس‌ها

- با فرض اینکه هر یک از ماتریس‌های A ، B و C را به چهار قسمت تقسیم کنیم، محاسبات به صورت زیر انجام می‌شود.

$$C = A \cdot B \Rightarrow \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

- بنابراین خواهیم داشت :

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

ضرب ماتریس‌ها

- بنابراین ضرب یک جفت ماتریس $n \times n$ را به ضرب هشت جفت ماتریس $n/2 \times n/2$ تبدیل کردیم.
- توجه کنید که در این محاسبات نتیجهٔ ضرب $A_{11} \cdot B_{11}$ و همچنین $A_{12} \cdot B_{21}$ باید در C_{11} ذخیره شود.

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

- حالت پایه در این الگوریتم وقتی است که می خواهیم دو ماتریس 1×1 را در هم ضرب کنیم که در این حالت در واقع دو عدد را در هم ضرب می‌کنیم.

ضرب ماتریس‌ها

- الگوریتم تقسیم و حل برای ضرب دو ماتریس را می‌توانیم به صورت زیر بنویسیم.

Algorithm Matrix

```
   function MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
    ▷ Base case.
1: if n==1 then
2:    c11 = c11 + a11 * b11
3:    return
    ▷ Divide.
4: Partition A, B, and C into n/2 × n/2 submatrices
   A11, A12, A21, A22; B11, B12, B21, B22;
   C11, C12, C21, C22; respectively
```

ضرب ماتریس‌ها

Algorithm Matrix

```
    ▷ Conquer.
 5: Matrix-Multiply-Recursive (A11, B11, C11, n/2)
 6: Matrix-Multiply-Recursive (A11, B12, C12, n/2)
 7: Matrix-Multiply-Recursive (A21, B11, C21, n/2)
 8: Matrix-Multiply-Recursive (A21, B12, C22, n/2)
 9: Matrix-Multiply-Recursive (A12, B21, C11, n/2)
10: Matrix-Multiply-Recursive (A12, B22, C12, n/2)
11: Matrix-Multiply-Recursive (A22, B21, C21, n/2)
12: Matrix-Multiply-Recursive (A22, B22, C22, n/2)
```

ضرب ماتریس‌ها

- یک ضرب ماتریسی با اندازهٔ n را به ۸ ضرب ماتریسی با اندازهٔ $n/۲$ تبدیل کردیم.
- فرض می‌کنیم که در تقسیم ماتریس به ماتریس‌های کوچک‌تر، تنها اندیس‌ها را نامگذاری مجدد می‌کنیم و بنابراین عملیات در زمان ثابت می‌تواند انجام شود.
- بنابراین برای تحلیل این الگوریتم، می‌توانیم از رابطه بازگشتی زیر استفاده کنیم.

$$T(n) = 8T(n/2) + \Theta(1)$$

ضرب ماتریس‌ها

- با حل کردن این معادله به دست می‌آوریم $T(n) = \Theta(n^3)$.
- بنابراین روش تقسیم و حل زمان محاسبات را در ضرب ماتریسی کاهش نمی‌دهد.
- با استفاده از الگوریتم استراسن [1] که از یک الگوریتم تقسیم و حل بهینه‌تر استفاده می‌کند، زمان محاسبات کاهش پیدا خواهد کرد.

[1] Strassen

الگوریتم استراسن

- ضرب دو ماتریس $n \times n$ را در زمان کمتر از n^3 نیز می‌توان انجام داد. از آنجایی که برای ضرب دو ماتریس مربعی با اندازهٔ n دقیقا به n^3 گام محاسباتی نیاز است، بسیاری بر این باور بودند که ضرب ماتریسی نمی‌تواند در زمانی کمتر صورت بگیرد تا اینکه در سال ۱۹۶۹ ولکر استراسن [1] ریاضیدان آلمانی، الگوریتمی با زمان اجرای $\Theta(n^{lg7})$ ابداع کرد. از آنجایی که $lg7 = 2.8073...$ ، بنابراین می‌توان گفت الگوریتم استراسن در زمان $O(n^{2.81})$ ضرب دو ماتریس را محاسبه می‌کند.
- الگوریتم استراسن یک الگوریتم از نوع و تقسیم و حل است.

[1] Volker Strassen

الگوریتم استراسن

- ایدهٔ الگوریتم استراسن این است که در مراحل تقسیم و ترکیب از عملیات بیشتری استفاده می‌کند و بنابراین مراحل تقسیم و ترکیب در این الگوریتم نسبت به مراحل تقسیم و ترکیب در الگوریتم تقسیم و حل عادی زمان بیشتری صرف می‌کند ولی در ازای این افزایش زمان، در مرحله حل بازگشتی زمان کمتری مصرف می‌شود. در واقع در مرحله بازگشتی به جای فراخوانی ۸ تابع بازگشتی ۷ تابع بازگشتی فراخوانی می‌شوند.
- به عبارت دیگر عملیات مورد نیاز برای یکی از فراخوانی‌های بازگشتی توسط تعدادی عملیات جمع در مراحل تقسیم و ترکیب انجام می‌شود.

الگوریتم استراسن

- به عنوان مثال فرض کنید می‌خواهیم به ازای دو عدد دلخواه x و y ، مقدار $x^2 - y^2$ را محاسبه کنیم. اگر بخواهیم این محاسبات را به صورت معمولی انجام دهیم، باید ابتدا x و y را به توان ۲ برسانیم و سپس دو مقدار به دست آمده را از هم کم کنیم. اما یک روش دیگر برای این محاسبات وجود دارد.
- می‌دانیم $x^2 - y^2 = (x + y)(x - y)$ ، بنابراین می‌توانیم این محاسبات را با یک عمل ضرب و دو عمل جمع و تفریق انجام دهیم. اگر x و y دو عدد باشند، زمان انجام محاسبات تفاوت چندانی نخواهد کرد، اما اگر x و y دو ماتریس بزرگ باشند، یک عمل ضرب کمتر بهبود زیادی در زمان اجرا ایجاد می‌کند.
- توجه کنید که جمع دو ماتریس مربعی با اندازهٔ n در زمان $O(n^2)$ انجام می‌شود، و ضرب دو ماتریس در زمان $O(n^3)$

الگوریتم استراسن

- حال که با ایدهٔ الگوریتم استراسن آشنا شدیم، الگوریتم را بررسی می‌کنیم.

۱. اگر $n = 1$ ، آنگاه هر ماتریس تنها یک درایه دارد. در این صورت باید یک عملیات ضرب ساده انجام داد که در زمان $\Theta(1)$ امکان پذیر است. اگر $n \neq 1$ ، آنگاه هر یک از ماتریس‌های ورودی A و B را به چهار ماتریس $n/2 \times n/2$ تقسیم می‌کنیم. این عملیات نیز در $\Theta(1)$ امکان پذیر است.

۲. با استفاده از زیر ماتریس‌های به دست آمده از مرحله قبل تعداد ۱۰ ماتریس $S_1, S_2, ..., S_{10}$ محاسبه می‌شوند. این عملیات در زمان $\Theta(n^2)$ انجام می‌شود.

۳. تابع ضرب ماتریسی به تعداد ۷ بار بر روی ماتریس‌های S_i، A_{ij}، B_{ij} که ابعاد هر کدام $n/2 \times n/2$ است، به طور بازگشتی انجام می‌شود. نتیجهٔ این محاسبات در ۷ ماتریس $P_1, P_2, ..., P_7$ ذخیره می‌شود. عملیات این مرحله در زمان $7T(n/2)$ انجام می‌شود.

۴. با استفاده از ماتریس‌های $P_1, P_2, ..., P_7$ ، ماتریس‌های $C_{11}, C_{12}, C_{21}, C_{22}$ محاسبه می‌شود. این عملیات نیز در زمان $\Theta(n^2)$ انجام می‌شود.

الگوریتم استراسن

- بنابراین زمان کل مورد نیاز برای الگوریتم استراسن از رابطه بازگشتی زیر به دست می‌آید.

$$T(n) = 7T(n/2) + \Theta(n^2)$$

- با حل این رابطه بازگشتی به دست می‌آوریم :

$$T(n) = \Theta(n^{\lg 7}) = O(n^{2.81})$$

الگوریتم استراسن

- حال ببینیم ماتریس‌های P_k چگونه با استفاده از ماتریس‌های A_{ij} و B_{ij} محاسبه می‌شوند.

الگوریتم استراسن

- در مرحلهٔ اول تعداد ۱۰ ماتریس S_i به صورت زیر محاسبه می‌شوند.

$$S_1 = B_{12} - B_{22}$$
$$S_2 = A_{11} + A_{12}$$
$$S_3 = A_{21} + A_{22}$$
$$S_4 = B_{21} - B_{11}$$
$$S_5 = A_{11} + A_{22}$$
$$S_6 = B_{11} + B_{22}$$
$$S_7 = A_{12} - A_{22}$$
$$S_8 = B_{21} + B_{22}$$
$$S_9 = A_{11} - A_{21}$$
$$S_{10} = B_{11} + B_{12}$$

الگوریتم استراسن

- در محاسبات فوق ۱۰ بار ماتریس‌های $n/2 \times n/2$ را با هم جمع کردیم که این عملیات در زمان $\Theta(n^2)$ امکان پذیر است.

الگوریتم استراسن

- در مرحلهٔ بعد ۷ ماتریس P_i را با استفاده از زیر ماتریس‌های A_{ij} و B_{ij} و ماتریس‌های S_i بدست می‌آوریم.

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 (= A_{11} \cdot B_{12} - A_{11} \cdot B_{22}) \\
P_2 &= S_2 \cdot B_{22} (= A_{11} \cdot B_{22} + A_{12} \cdot B_{22}) \\
P_3 &= S_3 \cdot B_{11} (= A_{21} \cdot B_{11} + A_{22} \cdot B_{11}) \\
P_4 &= A_{22} \cdot S_4 (= A_{22} \cdot B_{21} - A_{22} \cdot B_{11}) \\
P_5 &= S_5 \cdot S_6 (= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}) \\
P_6 &= S_7 \cdot S_8 (= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}) \\
P_7 &= S_9 \cdot S_{10} (= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12})
\end{aligned}
$$

الگوریتم استراسن

- بنابراین در اینجا به ۷ عملیات ضرب نیاز داریم که به صورت بازگشتی انجام می‌شوند.
- در مرحلهٔ آخر باید زیر ماتریس‌های C_{ij} را با استفاده از ماتریس‌های P_i به دست آوریم.
- این محاسبات به صورت زیر انجام می‌شوند.

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_5 + P_1 - P_3 - P_7$$

الگوریتم استراسن

- با بسط دادن این روابط می‌توانیم C_{ij} ها را بر اساس A_{ij} و B_{ij} ها به دست آوریم و نشان دهیم که عملیات ضرب به درستی انجام می‌شود.

$$
\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 (= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}) \\
C_{12} &= P_1 + P_2 (= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}) \\
C_{21} &= P_3 + P_4 (= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}) \\
C_{22} &= P_5 + P_1 - P_3 - P_7 (= A_{21} \cdot B_{12} + A_{22} \cdot B_{22})
\end{aligned}
$$

- در مرحله آخر تنها از عملیات جمع استفاده می‌کنیم بنابراین محاسبهٔ C_{ij} ها در زمان $\Theta(n^2)$ انجام می‌پذیرد.

نزدیک‌ترین زوج نقطه

- مسئلهٔ یافتن نزدیک‌ترین زوج نقطه [1] در بین تعدادی نقاط در صفحه یکی دیگر از مسائلی است که با استفاده از روش تقسیم و حل قابل حل است.
- این مسئله در حوزه‌های متعددی کاربرد دارد. به طور کلی هر موجودی را می‌توان با استفاده از تعدادی ویژگی مدلسازی کرد. این ویژگی‌ها یک نقطه در یک فضای چند بعدی ایجاد می‌کنند. حال برای پیدا کردن دو موجود با ویژگی‌های مشابه باید در فضای چند بعدی تشکیل شده، به دنبال نزدیک‌ترین زوج نقطه‌ها بگردیم.
- مسئلهٔ یافتن نزدیک‌ترین زوج نقطه را ابتدا در فضای یک بعدی و سپس در فضای دو بعدی بررسی می‌کنیم.

[1] closest pair of points

نزدیک‌ترین زوج نقطه

- فرض کنید مجموعهٔ S حاوی n نقطه در فضای یک بعدی بر روی یک محور داده شده است. می‌خواهیم از بین این n نقطه دو نقطه پیدا کنیم که فاصلهٔ آن دو به یکدیگر از فاصلهٔ هر زوج نقطهٔ دیگری کمتر باشد. به عبارت دیگر می‌خواهیم نزدیک‌ترین زوج نقطه را در مجموعهٔ S پیدا کنیم.
- در یک الگوریتم ابتدایی باید فاصلهٔ بین همهٔ زوج نقطه‌ها را محاسبه کنیم و از بین فاصله‌های محاسبه شده، کوتاهترین فاصله و نقاط متناظر با آن فاصله را پیدا کنیم. از آنجایی که تعداد n^2 زوج نقطه وجود دارد، زمان مورد نیاز برای محاسبهٔ همهٔ فاصله‌ها و حل مسئله $O(n^2)$ است.

نزدیک‌ترین زوج نقطه

- با استفاده از روش تقسیم و حل، این مسئله را در زمان کمتری حل می‌کنیم.
- تقسیم : نقاط مجموعه S را بر روی محور مختصات x مرتب می‌کنیم. سپس مجموعهٔ S حاوی n نقطه را به دو مجموعه S_1 و S_2 هرکدام شامل $n/2$ نقطه تقسیم می‌کنیم.
- حل : الگوریتم را به طور بازگشتی برای مجموعهٔ S_1 و S_2 فراخوانی می‌کنیم.
- ترکیب : نزدیک‌ترین زوج نقطه در بین نقاط مجموعهٔ S یا در مجموعهٔ S_1 است و یا S_2 و یا زوج نقطه‌ای که از اولین نقطه S_2 و آخرین نقطهٔ S_1 تشکیل شده است. پس در بین این سه حالت باید نزدیک‌ترین زوج نقطه محاسبه و بازگردانده شود.
- حالت پایه : در حالت پایه مجموعهٔ S شامل ۲ نقطه است که فاصلهٔ بین این دو نقطه بازگردانده می‌شود.

نزدیک‌ترین زوج نقطه

- این الگوریتم را می‌توانیم به صورت زیر بنویسیم.

Algorithm Closest Pair of One-Dimensional Points

```
   function CLOSEST-PAIR-1D(S)  ▷ S is already sorted
1: if |S| == 2 then
2:    return d = p2 - p1
3: Divide S from the mid-point and conxtruct S1 and S2
4: d1 = Closest-Pair-1D(S1)
5: d2 = Closest-Pair-1D(S2)
6: p1 = max(S1)
7: p2 = min(S2)
8: d = min {d1, d2, p2-p1}
9: return d
```

نزدیک‌ترین زوج نقطه

- برای محاسبهٔ زمان اجرای این الگوریتم، زمان اجرای مرحله تقسیم و ترکیب را محاسبه می‌کنیم که هر دو در زمان ثابت انجام می‌شوند. برای محاسبه زمان اجرا از رابطهٔ بازگشتی $T(n) = 2T(n/2) + O(1)$ استفاده می‌کنیم، بنابراین به دست می‌آوریم $T(n) = O(n \lg n)$.

نزدیک‌ترین زوج نقطه

- الگوریتم یافتن نزدیک‌ترین زوج نقطه در فضای دو بعدی به طور مشابه با الگوریتم یافتن نزدیک‌ترین زوج نقطه در فضای یک بعدی عمل می‌کند.
- الگوریتم تقسیم و حل به صورت زیر عمل می‌کند.
- تقسیم : در مرحله تقسیم یک خط عمودی در فضای دو بعدی رسم می‌کنیم که مجموعهٔ S را از وسط به دو زیر مجموعهٔ S_1 و S_2 تقسیم می‌کند.
- حل : نزدیک‌ترین زوج نقطه را در بین نقاط مجموعهٔ S_1 و S_2 به طور بازگشتی پیدا می‌کنیم. فرض کنیم فاصلهٔ نزدیک‌ترین زوج نقطه در مجموعهٔ S_1 برابر با d_1 و در مجموعهٔ S_2 برابر با d_2 باشد.
- ترکیب : فاصلهٔ نزدیک‌ترین زوج نقطه در مجموعهٔ S یا برابر است با d_1 یا d_2 و یا برابر است با فاصله یکی از زوج نقطه‌ها که یکی از آنها در S_1 و دیگری در S_2 است.

نزدیک‌ترین زوج نقطه

- برای پیدا کردن فاصلهٔ زوج نقطه‌هایی که یکی از آنها در S_1 و دیگری در S_2 است، سه خط عمودی را در نظر بگیرید. خط عمودی اول طوری رسم شده است که دقیقا وسط مجموعهٔ نقاط قرارگرفته است یعنی فاصلهٔ آن از نقطه‌ای که مقدار x آن کمترین است برابر است با فاصله آن خط از نقطه‌ای که مقدار x آن بیشترین است. سپس دو خط موازی دیگر با این خط وسط در نظر بگیرید که فاصله هر کدام از آن‌ها از خط وسط برابر با $d = min(d_1, d_2)$ باشد. بدین ترتیب یک نوار با عرض $2d$ تشکیل داده‌ایم.

- توجه کنید هیچ زوج نقطه‌ای که خارج از نوار و یکی از آنها در S_1 و دیگری در S_2 است، وجود نخواهد داشت که فاصلهٔ آنها از d کمتر باشد. بنابراین باید بررسی کنیم آیا در این نوار زوج نقطه‌ای وجود دارد که فاصله‌اش از d کمتر است یا خیر.

- می‌توان به طور هندسی اثبات کرد که برای این کار کافی است از بالاترین نقطه شروع کنیم و فاصلهٔ هر نقطه را با ۷ نقطهٔ بعدی مقایسه کنیم که این کار در زمان $O(n)$ امکان پذیر است.

- بنابراین زمان اجرای این الگوریتم را می‌توانیم از رابطه $T(n) = 2T(n/2) + O(n)$ محاسبه کنیم که به دست می‌آوریم $T(n) = O(nlgn)$.

نزدیک‌ترین زوج نقطه

- برای اینکه نشان دهیم هر نقطه را در نوار میانی حداکثر با ۷ نقطه باید مقایسه کنیم، ابتدا نوار میانی را به مربع‌هایی با ابعاد $n/2 \times n/2$ تقسیم می‌کنیم.
- سپس نشان می‌دهیم که در یک مربع با ابعاد $n/2 \times n/2$ نمی‌تواند بیش از یک نقطه وجود داشته باشد، چرا که وجود بیش از دو نقطه در مربعی با این ابعاد نشان دهندهٔ این است که دو نقطه در یک طرف مرز میانی فاصله ای کمتر از d دارند و این با توجه به مفروضات غیرممکن است.
- سپس نشان می‌دهیم در بدترین حالت تعداد نقاطی که در هر یک از مربع‌های کوچک قرار دارند و فاصلهٔ آنها از یک نقطه معین می‌تواند کمتر از d باشد، ۷ است.

حل روابط بازگشتی

- در مسائل تقسیم و حل دیدیم چگونه می‌توان از روابط بازگشتی برای محاسبهٔ زمان اجرای الگوریتم‌ها بهره گرفت. در اینجا چند روش برای حل روابط بازگشتی مطرح می‌کنیم که عبارتند از روش جایگذاری [1] ، روش درخت بازگشت [2] و روش قضیه اصلی [3] .

[1] substitution method
[2] recursion-tree method
[3] master theorem method

روش جایگذاری

- روش جایگذاری برای حل روابط بازگشتی از دو گام تشکیل شده است. در گام اول جواب رابطهٔ بازگشتی یا عبارت فرم بسته [1] که در رابطهٔ بازگشتی صدق می‌کند حدس زده می‌شود. در گام دوم توسط استقرای ریاضی [2] اثبات می‌شود که جوابی که حدس زده شده است درست است و در رابطهٔ بازگشتی صدق می‌کند.
- برای اثبات توسط استقرای ریاضی، ابتدا باید ثابت کرد که جواب حدس زده شده برای مقادیر کوچک n درست است. سپس باید اثبات کرد که اگر جواب حدس زده شده برای n درست باشد، برای n+۱ نیز درست است. در این روش از جایگذاری جواب حدس زده شده در رابطهٔ اصلی برای اثبات استفاده می‌شود و به همین دلیل روش جایگذاری نامیده می‌شود.
- متاسفانه هیچ قاعدهٔ کلی برای حدس زدن جواب رابطهٔ بازگشتی وجود ندارد و یک حدس خوب به کمی تجربه و خلاقیت نیاز دارد.

[1] closed-form expression
[2] mathematical induction

روش جایگذاری

- برای مثال فرض کنید می‌خواهیم رابطهٔ $T(n) = 2T(n-1)$ را حل کنیم.
- این رابطه را برای n های کوچک می‌نویسیم و حدس می‌زنیم $T(n) = 2^n$ باشد.
- سپس رابطه را با استفاده از استقرا اثبات می‌کنیم.

روش جایگذاری

- در برخی مواقع یک رابطهٔ بازگشتی شبیه رابطه‌هایی است که جواب آنها را می‌دانیم و در چنین مواقعی می‌توانیم از حدس استفاده کنیم.
- برای مثال رابطه $T(n) = 2T(n/2 + 17) + \Theta(n)$ را در نظر بگیرید. شبیه این رابطه را بدون عدد ۱۷ قبلا دیده‌ایم اما می‌توانیم حدس بزنیم که این عدد برای n های بزرگ تأثیر زیادی ندارد. پس حدس میزنیم که جواب این رابطه $T(n) = O(n \lg n)$ باشد.
- یک روش دیگر برای حدس زدن این است که ابتدا یک کران پایین حدس زده و سپس کران پایین را افزایش دهیم تا به جواب واقعی نزدیک شویم.

روش درخت بازگشت

- روش دیگر برای حل مسائل بازگشتی، استفاده از درخت بازگشت [1] است.
- در این روش هر رأس از درخت، هزینه محاسبات یکی از زیر مسئله‌ها را نشان می‌دهد.
- هزینهٔ کل اجرای یک برنامه عبارت است از هزینه‌ای که در سطح صفر درخت برای تقسیم و ترکیب نیاز است به علاوه هزینه محاسبه زیر مسئله‌ها. به همین ترتیب هزینه محاسبه هر یک از زیر مسئله‌های سطح اول تشکیل می‌شود از هزینه تقسیم و ترکیب به علاوهٔ هزینهٔ زیر مسئله‌های سطح دوم و به همین ترتیب الی آخر.
- بنابراین اگر هزینهٔ محاسبهٔ همه رئوس درخت بازگشت را جمع کنیم، هزینه کل اجرای برنامه به دست می‌آید.

[1] recursion tree

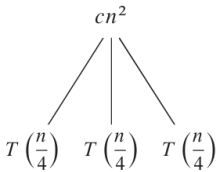
روش درخت بازگشت

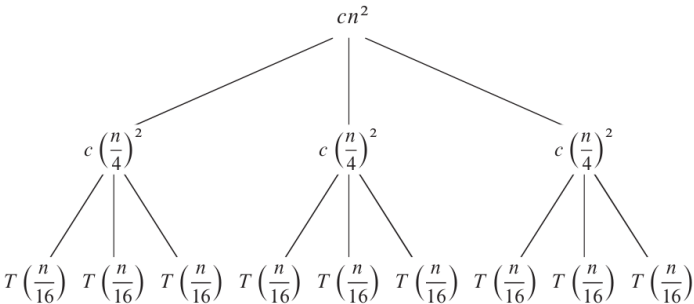
- یک مثال درخت بازگشت را در اینجا بررسی می‌کنیم. رابطهٔ بازگشتی $T(n) = 3T(n/4) + \Theta(n^2)$ را در نظر بگیرید.

روش درخت بازگشت

- شکل زیر تشکیل درخت بازگشت را برای این رابطه بازگشتی در دو مرحله اول نشان می‌دهد.

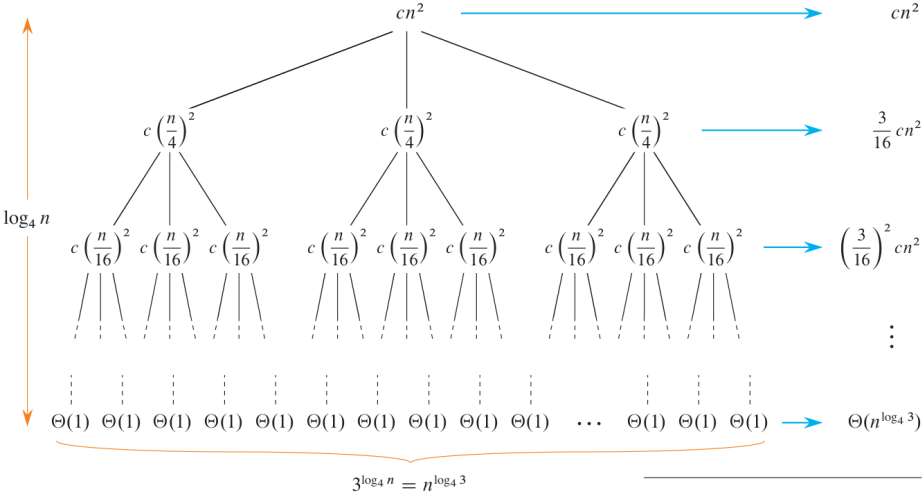
$T(n)$





روش درخت بازگشت

- اگر مجموع هرینه‌ها را در سطح محاسبه کنیم، درختی با هزینه‌های قید شده در زیر خواهیم داشت.



روش درخت بازگشت

- سپس هزینه‌های سطوح این درخت بازگشت را با هم جمع می‌کنیم و جواب رابطه بازگشتی را به دست می‌آوریم.

روش درخت بازگشت

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) \qquad\qquad (\Theta(n^{\log_4 3}) = O(n^{0.8}) = O(n^2)).
\end{aligned}
$$

روش قضیه اصلی

- روش قضیه اصلی [1] برای حل مسائل بازگشتی استفاده می‌شود که به صورت $T(n) = aT(n/b) + f(n)$ هستند به طوری که $a > 0$ و $b > 1$ دو ثابت هستند.
- تابع $f(n)$ در اینجا تابع محرک [2] نامیده می‌شود و یک رابطهٔ بازگشتی که به شکل مذکور است، رابطهٔ بازگشتی اصلی [3] نامیده می‌شود.
- در واقع رابطهٔ بازگشتی اصلی زمان اجرای الگوریتم‌های تقسیم و حل را توصیف می‌کند که مسئله‌ای به اندازهٔ n را به a زیر مسئله هر کدام با اندازهٔ n/b تقسیم می‌کنند. تابع $f(n)$ هزینه تقسیم مسئله به زیر مسئله‌ها به علاوه هزینه ترکیب زیر مسئله‌ها را نشان می‌دهد.
- اگر یک رابطهٔ بازگشتی شبیه رابطه قضیه اصلی باشد و علاوه بر آن چند عملگر کف و سقف در آن وجود داشته باشد، همچنان می‌توان از رابطهٔ قضیه اصلی استفاده کرد.

[1] master theorem method
[2] driving function
[3] master recurrence

روش قضیه اصلی

- قضیه اصلی : فرض کنید $a > 0$ و $b > 1$ دو ثابت باشند و $f(n)$ یک تابع باشد که برای اعداد بسیار بزرگ تعریف شده باشد.
- رابطهٔ بازگشتی $T(n)$ که بر روی اعداد طبیعی $n \in \mathbb{N}$ تعریف شده است را به صورت زیر در نظر بگیرید.

$$T(n) = aT(n/b) + f(n)$$

روش قضیه اصلی

- در این‌صورت رفتار مجانبی $T(n)$ به صورت زیر است :

۱- اگر ثابت $\epsilon > 0$ وجود داشته باشد به طوری‌که $f(n) = O(n^{\log_b^a - \epsilon})$ آنگاه $T(n) = \Theta(n^{\log_b^a})$.

۲- اگر ثابت $k \geqslant 0$ وجود داشته باشد به طوری‌که $f(n) = \Theta(n^{\log_b^a} \lg^k n)$ آنگاه
$T(n) = \Theta(n^{\log_b^a} \lg^{k+1} n)$.

۳- اگر ثابت $\epsilon > 0$ وجود داشته باشد به طوری‌که $f(n) = \Omega(n^{\log_b^a + \epsilon})$ و اگر $f(n)$ در رابطهٔ
$af(n/b) \leqslant cf(n)$ صدق کند به ازای $c < 1$ و n های به اندازهٔ کافی بزرگ، آنگاه $T(n) = \Theta(f(n))$.

روش قضیه اصلی

- رابطهٔ بازگشتی $T(n) = 9T(n/3) + n$ را در نظر بگیرید. در این رابطه داریم $a = 9$ و $b = 3$ بنابراین $n^{\log_b^a} = n^{\log_3^9} = \Theta(n^2)$. از آنجایی که $f(n) = n = O(n^{2-\epsilon})$ به ازای هر ثابت $\epsilon \leqslant 1$ بنابراین می‌توانیم حالت اول در قضیه اصلی را در نظر بگیریم و نتیجه بگیریم $T(n) = \Theta(n^2)$.

روش قضیه اصلی

- رابطهٔ بازگشتی $T(n) = T(2n/3) + 1$ را در نظر بگیرید. در این رابطه داریم $a = 1$ و $b = 3/2$ بنابراین $n^{\log_b^a} = n^{\log_{3/2}^1} = n^0 = 1$. در اینجا حالت دوم در قضیه اصلی را داریم یعنی $f(n) = 1 = \Theta(n^{\log_b^a} lg^0 n) = \Theta(1)$ بنابراین جواب رابطهٔ بازگشتی برابر است با $T(n) = \Theta(lgn)$.

روش قضیه اصلی

- در رابطهٔ بازگشتی $T(n) = 3T(n/4) + n\lg n$ داریم $a = 3$ و $b = 4$ که بدین معنی است که $n^{\log_b^a} = n^{\log_4^3} = \Theta(n^{0.793})$. از آنجایی که $f(n) = n\lg n = \Omega(n^{\log_4^3 + \epsilon})$ جایی که ϵ حدود 0.2 است، بنابراین حالت سوم در قضیه اصلی را می‌توانیم در نظر بگیریم اگر شرط $af(n/b) \leqslant cf(n)$ برقرار باشد.

$af(n/b) = 3(n/4)\lg(n/4) \leqslant (3/4)n\lg n = 3/4f(n)$

بنابراین با استفاده از حالت سوم جواب رابطهٔ بازگشتی برابراست با $T(n) = \Theta(n\lg n)$.

روش قضیه اصلی

- رابطهٔ بازگشتی $T(n) = 2T(n/2) + \Theta(n)$ رابطه‌ای بود که برای مرتب‌سازی ادغامی به دست آوردیم. از آنجایی که $a = 2$ و $b = 2$ داریم $n^{\log_2^2} = n$. حالت دوم در اینجا برقراراست زیرا به ازای $k = 0$ داریم $f(n) = \Theta(n)$ و بنابراین جواب رابطهٔ بازگشتی برابر است با $T(n) = \Theta(n \lg n)$.

روش قضیه اصلی

- رابطهٔ $T(n) = 8T(n/2) + \Theta(1)$ زمان اجرای الگوریتم ضرب ماتریسی را توصیف می‌کند. در اینجا داریم $a = 8$ و $b = 2$ بنابراین $n^{\log_2^8} = n^3$. تابع محرک $f(n) = \Theta(1)$ است و بنابراین به ازای هر $\epsilon < 3$ داریم $f(n) = O(n^{3-\epsilon})$. بنابراین حالت اول قضیه اصلی برقرار است. نتیجه می‌گیریم $T(n) = \Theta(n^3)$.

روش قضیه اصلی

- در تحلیل زمان اجرای الگوریتم استراسن رابطهٔ $T(n) = 7T(n/2) + \Theta(n^2)$ را به دست آوردیم. در این رابطهٔ بازگشتی $a = 7$ و $b = 2$ بنابراین $n^{\log_2^7} = n^{lg7}$. از آنجایی که $lg7 = 2.8073...$ ، می‌توانیم قرار دهیم $\epsilon = 0.8$ و برای تابع محرک خواهیم داشت $f(n) = \Theta(n^2) = O(n^{lg7-\epsilon})$ ، پس حالت اول در قضیه اصلی برقرار است و بنابراین جواب رابطهٔ بازگشتی برابر است با $T(n) = \Theta(n^{lg7})$.