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 $P(n+1)$ نیز درست است (گام استقرا).

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

طراحی الگوریتم با استقرا

- برای مثال با استفاده از استقرا می‌توان اثبات کرد:

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

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

طراحی الگوریتم با استقرا

- اثبات :
- پایه استقرا درست است زیرا $P(1) = 1 = \frac{1(2)}{2} = 1$
- 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 $P(n+1) = \frac{(n+1)(n+2)}{2}$. بنابراین گام استقرا نیز درست است.
- با استفاده از این رابطه برای محاسبهٔ n عدد کافی است از رابطه $P(n)$ استفاده کنیم. این الگوریتم در زمان $O(1)$ انجام می‌شود، در حالی که جمع n عدد با استفاده از یک حلقه در زمان $O(n)$ انجام می‌شود.

طراحی الگوریتم با استقرا

- استقرای ریاضی براساس اصول دومینو [1] است. فرض کنید تعداد زیادی دومینو به صورت ایستاده درکنار یکدیگر قرارگرفته‌اند و می‌خواهیم همهٔ دومینوهای ایستاده را بیاندازیم. برای اینکه همهٔ دومینوها بر زمین بیفتند کافی است دومینوها به گونه‌ای قرار داده شوند که با افتادن اولین دومینو، دومین دومینو برزمین بیافتد و با افتادن دومی، سومی و به همین ترتیب با افتادن n امین دومینو، $n+1$ امین دومینو بر زمین بیافتد. سپس کافی است به اولین دومینو ضربه‌ای بزنیم تا همه دومینوهای ایستاده بیافتند و نیازی به انداختن تک‌تک آنها نداریم.

[1] domino principle

طراحی الگوریتم با استقرا

- استقرای ریاضی در طراحی الگوریتم‌ها بسیار پر استفاده است.
- برای طراحی یک الگوریتم برای حل یک مسئله با استفاده از استقرا کافی است :

۱. مسئله را در حالت پایه یعنی حالتی که اندازه ورودی کوچک است حل کنیم.

۲. نشان‌دهیم چگونه می‌توان یک مسئله را با استفاده از یک زیر مسئله (یعنی مسئله‌ای با اندازهٔ کوچک‌تر) حل کرد.

طراحی الگوریتم با استقرا

- فرض کنید می‌خواهیم به ازای دنباله‌ای از اعداد حقیقی a_0 ، a_1 ، a_2 ، $\cdots$ ، a_n و عدد داده شده x، مقدار چند جمله‌ای زیر ار محاسبه کنیم.

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

طراحی الگوریتم با استقرا

- یک الگوریتم بدیهی برای حل این مسئله با جایگذاری اعداد a_i و x در چند جمله $P_n(x)$ مقدار آن را محاسبه می‌کند.

Algorithm Compute Polynomial

```
function ComputePolynomial(a[], x)
P = a[0]
for i = 1 to n do
    X = 1
    for j = 1 to i do
        X = X * x
    P = P + a[i] * X
return P
```

- پیچیدگی زمانی این الگوریتم $O(n^2)$ است.

- حال می‌خواهیم با استفاده از استقرا این مسئله را در زمان کمتری حل کنیم.

طراحی الگوریتم با استقرا

- برای حل مسئله با استفاده از استقرا باید بتوانیم مسئله را بر اساس یک زیر مسئله بیان کنیم.
- یک زیر مسئله از مسئلهٔ محاسبه چند جمله‌ای را به صورت زیر در نظر بگیرید.

$$P_{n-1}(x) = a_n x^{n-1} + a_{n-1} x^{x-2} + \cdots + a_1$$

- فرض کنید جواب $P_{n-1}(x)$ داده شده است. چگونه می‌توانیم $P_n(x)$ را محاسبه کنیم؟

طراحی الگوریتم با استقرا

- برای محاسبه $P_n(x)$ می‌توانیم رابطه‌ای به صورت زیر بنویسیم.

$$P_n(x) = x \cdot P_{n-1}(x) + a_0$$

- همچنین می‌توانیم $P_{n-1}(x)$ را بر اساس $P_{n-2}(x)$ محاسبه کنیم.
- داریم :

$$P_{n-2}(x) = a_n x^{n-2} + a_{n-1} x^{n-3} + \cdots + a_2$$

- بنابراین خواهیم داشت :

$$P_{n-1}(x) = x \cdot P_{n-2}(x) + a_1$$

طراحی الگوریتم با استقرا

- در حالت کلی برای محاسبه $P_{n-j}(x)$ با استفاده از یک زیرمسئله می‌توانیم رابطهٔ زیر را ارائه کنیم:

$$P_{n-j}(x) = x \cdot P_{n-(j+1)}(x) + a_j$$

- در حالت پایه داریم:

$$P_0(x) = a_n$$

طراحی الگوریتم با استقرا

- فرض کنیم $n - j = i$ ، در اینصورت خواهیم داشت :

$$\begin{cases} P_i(x) = x \cdot P_{i-1}(x) + a_{n-i} & \text{اگر } i > 0 \\ P_0(x) = a_n & \text{اگر } i = 0 \end{cases}$$

طراحی الگوریتم با استقرا

- بنابراین با استفاده از رابطه بازگشتی به دست آمده می‌توانیم الگوریتمی به صورت زیر بنویسیم.

Algorithm Compute Polynomial

```
function ComputePolynomial(a[], x)
P = a[n]
for i = 1 to n do
    P = x * P + a[n-i]
return P
```

- پیچیدگی زمانی این الگوریتم $O(n)$ است که از الگوریتم بدیهی که در زمان $O(n^2)$ چند جمله‌ای را محاسبه می‌کند سریع‌تر است.

طراحی الگوریتم با استقرا

- این الگوریتم به روش هورنر [1] معروف است که توسط ریاضی‌دان انگلیسی ویلیام هورنر [2] ابداع شده است، گرچه خود هورنر آن را به ریاضی‌دان فرانسوی-ایتالیایی ژوزف لاگرانژ [3] نسبت داده است. گفته می‌شود این الگوریتم قبل از لاگرانژ احتمالاً توسط ریاضی‌دانان ایرانی و چینی ابداع شده است.

$$a_0 + a_1x + a_2x^2 + \cdots + a_nx^n = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + xa_n)\cdots)))$$

[1] Horner's method
[2] William Horner
[3] Joseph-Louis Lagrange

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] (تقسیم و غلبه) 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 `1` 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)
if p >= r then  ▷ zero or one element?
    return
q = ⌊ (p+r)/2 ⌋  ▷  midpoint of A[p:r]
Merge-Sort (A, p, q)  ▷  recursively sort A[p:q]
Merge-Sort (A, q+1, r)  ▷  recursively sort A[q+1:r]
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)
nl = q - p + 1  ▷ length of A[p:q]
nr = r - q  ▷ length of A[q+1 : r]
let L[ 0 : nl - 1 ] and R[ 0 : nr - 1 ] be new arrays
for i = 0 to nl - 1 do  ▷ copy A[p:q] into L[0:nl - 1]
    L[i] = A[p+i]
for j = 0 to nr - 1 do  ▷ copy A[q+1:r] into L[0:nr - 1]
    R[j] = A[q + j + 1]
```

Algorithm Merge Sort

```
function MERGE(A, p, q, r)
i = 0  ▷ i indexes the smallest remaining element in L
j = 0  ▷ j indexes the smallest remaining element in R
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].
while  i < nl and j < nr do
    if L[i] <= R[j] then
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
    k = k + 1
```

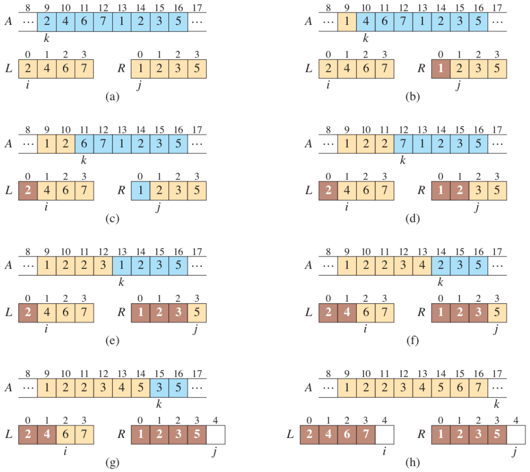
مرتب‌سازی ادغامی

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]
while i < nl do
    A[k] = L[i]
    i = i + 1
    k = k + 1
while  j < nr do
    A[k] = R[j]
    j = j + 1
    k = k + 1
```

مرتب‌سازی ادغامی

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

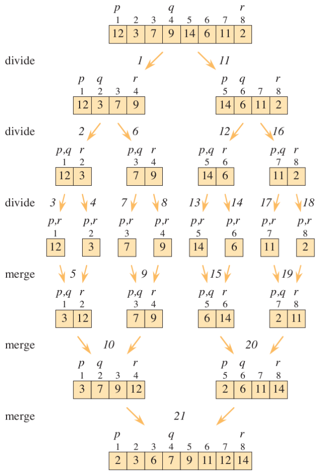


مرتب‌سازی ادغامی

- در حلقهٔ تکرار الگوریتم ادغام، در هر تکرار یکی از عناصر در آرایهٔ A کپی می‌شوند و در کل 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.

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

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 تقسیم n بر ۲ یک عدد صحیح به دست می‌آید.

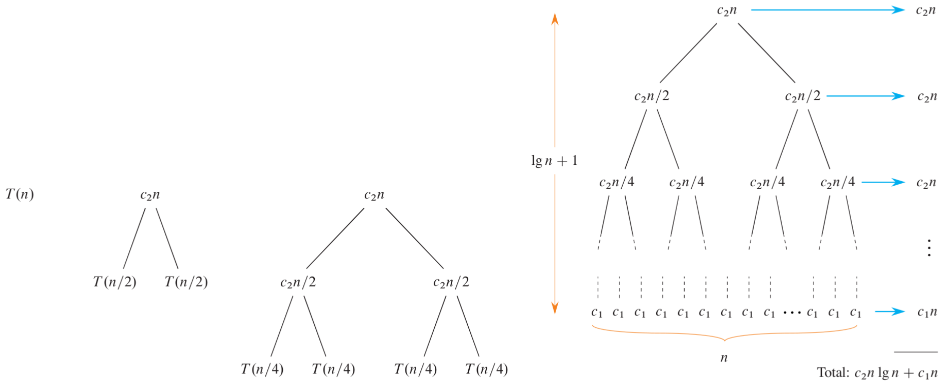
- زمان اجرای الگوریتم را به صورت زیر می‌نویسیم.

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

- 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

- سطح آخر، یعنی سطحی که برگ‌های درخت در آن قرار دارد، 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 $T(n) = \Theta(n \lg n)$.

جستجوی دودویی

- برای جستجوی یک مقدار در یک آرایه باید همهٔ عناصر آرایه را یک‌به‌یک بررسی کنیم. این جستجو برای یک آرایه با n عنصر در زمان $O(n)$ انجام می‌شود.
- حال فرض می‌کنیم می‌خواهیم یک مقدار را در یک آرایه مرتب شده پیدا کنیم.
- برای این کار می‌توانیم از یک الگوریتم تقسیم و حل به نام جستجوی دودویی [1] استفاده کنیم.

[1] binary search

جستجوی دودویی

- الگوریتم تقسیم و حل آرایه را به دو قسمت تقسیم می‌کند. برای جستجوی مقدار x در آرایه A ، ابتدا مقدار x با عنصر وسط آرایه یعنی A[n/2] مقایسه می‌شود. اگر x برابر با مقدار وسط آرایه بود، مقدار مورد نظر یافته شده است. اگر x کوچکتر از عنصر وسط آرایه بود، باید x را در نیمه اول آرایه یعنی A[1:n/2-1] جستجو کنیم. در غیراینصورت باید x را در نیمه دوم آرایه یعنی A[n/2+1:n] جستجو کنیم. این روند را برای زیر آرایه‌ها ادامه می‌دهیم تا یا x یافته شود یا مشخص شود که x در آرایه وجود ندارد.

جستجوی دودویی

- بنابراین مراحل انجام جستجوی دودویی به صورت زیر است.

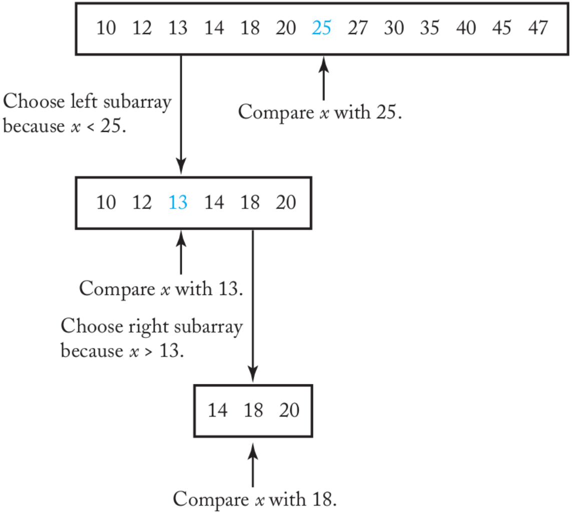
۱. تقسیم : برای پیدا کردن مقدار `x` در آرایه `A[low:high]` قرار می‌دهیم `mid=⌊(low + high)/2⌋`. اگر `A[mid]` برابر با `x` بود به نتیجه رسیده‌ایم در غیراینصورت آرایه را به دو قسمت `A[low:mid-1]` و `A[mid+1:high]` تقسیم می‌کنیم. این تقسیم تنها در صورتی می‌تواند انجام شود که `high` از `low` بزرگ‌تر باشد.

۲. حل : در صورتی که مقدار `x` از `A[mid]` کوچکتر بود، الگوریتم جستجو برای `A[low:mid-1]` فراخوانی می‌شود، در غیراینصورت برای `A[mid+1:high]` فراخوانی می‌شود.

۳. ترکیب : در گام ترکیب هیچ عملیاتی انجام نمی‌شود.

جستجوی دودویی

- برای پیدا کردن عدد ۱۸ در آرایهٔ زیر، الگوریتم به صورت زیر عمل می‌کند.



جستجوی دودویی

- الگوریتم جستجوی دودویی به صورت زیر است.

Algorithm Binary Search

```
function BinarySearch(A, x, low, high)
if (low > high) then
    return -1
mid = ⌊(low + high)/2⌋
if (x == A[mid]) then
    return mid
if (x < A[mid]) then
    return BinarySearch (A, x, low, mid-1)
else
    return BinarySearch (A, x, mid+1, high)
```

- برای جستجوی مقدار x جستجوی دودویی باید به صورت `BinarySearch(A, x, 1, n)` فراخوانی شود.

جستجوی دودویی

- در تقسیم یک آرایه به دو قسمت صرفا یک عملیات تقسیم انجام می‌شود. بنابراین $D(n) = O(1)$. تقسیم آرایه در زمان ثابت انجام می‌شود.
- بنابراین زمان اجرای الگوریتم جستجوی دودویی برای آرایه با n عنصر برابر است با زمان اجرای الگوریتم برای آرایه‌ای با $n/2$ عنصر به علاوه یک زمان ثابت.
- می‌توانیم بنویسیم $T(n) = T(\frac{n}{2}) + O(1)$ و $T(1) = 1$
- با حل این رابطه بازگشتی به دست می‌آوریم $T(n) = O(\lg n)$.

مرتب‌سازی سریع

- یکی از الگوریتم‌های مرتب‌سازی بسیار پر استفاده الگوریتم مرتب‌سازی سریع [1] است.
- این الگوریتم یک الگوریتم تقسیم و حل است. زمان اجرای آن در بدترین حالت $\Theta(n^2)$ است، اما در حالت میانگین در زمان $\Theta(n lg n)$ اجرا می‌شود. این الگوریتم به حافظه اضافی نیاز ندارد.

[1] quicksort algorithm

مرتب‌سازی سریع

- برای مرتب‌سازی آرایه `A[p:r]` این الگوریتم از روش تقسیم و حل به صورت زیر استفاده می‌کند.

۱. تقسیم : آرایهٔ `A[p:r]` به دو قسمت `A[p:q-1]` (قسمت پایین) [1] و `A[q+1:r]` (قسمت بالا) [2] تقسیم می‌شود به طوری که همه عناصر قسمت پایین از عنصر `A[q]` (عنصر محور) [3] کوچکتر یا برابرند و عناصر قسمت بالا از عنصر محور بزرگ‌ترند.

۲. حل : الگوریتم مرتب‌سازی سریع برای دو زیر آرایه `A[p:q-1]` و `A[q+1:r]` فراخوانی می‌شود.

۳. ترکیب : در این قسمت هیچ عملیاتی انجام نمی‌شود. از آنجایی که همه عناصر `A[p:q-1]` مرتب شده و کوچکتر یا مساوی `A[q]` هستند و همهٔ عناصر `A[q+1:r]` مرتب شده و بزرگتر از `A[q]` هستند، بنابراین کل آرایه `A[p:r]` مرتب شده است.

---
[1] low side
[2] high side
[3] pivot

مرتب‌سازی سریع

- الگوریتم مرتب‌سازی سریع به صورت زیر است.

Algorithm Quicksort

```
function QUICKSORT(A, p, r)
if p < r then  ▷ Partition the subarray around the pivot, which ends up in A[q].
    q = Partition (A, p, r)
    Quicksort (A, p, q-1)     ▷ recursively sort the low side
    Quicksort (A, q+1, r)     ▷ recursively sort the high side
```

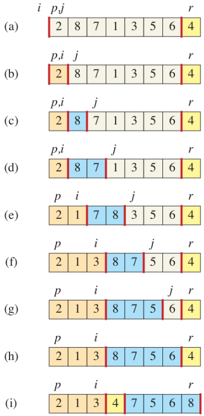
مرتب‌سازی سریع

- الگوریتم تقسیم‌بندی [1] باید عناصر آرایه را به گونه‌ای جابجا کند که همهٔ عناصر قسمت پایین از عنصر محور کوچک‌تر یا مساوی و عناصر قسمت بالا از عنصر محور بزرگ‌تر باشند.

[1] partition

مرتب‌سازی سریع

- در شکل زیر نحوه اجرای الگوریتم تقسیم‌بندی نشان داده شده است. عنصر محور در اینجا برابر است با `A[r]`.



مرتب‌سازی سریع

- الگوریتم تقسیم بندی به صورت زیر است.

Algorithm Partition

```
function PARTITION(A, p, r)
x = A[r]   ▷ the pivot
i = p - 1   ▷ highest index into the low side
for j=p to r-1 do   ▷ process each element other than the pivot
    if A[j] <= x then      ▷ does this element belong on the low side?
        i = i+1         ▷ index of a new slot in the low side
        exchange A[i] with A[j]         ▷ put this element there
exchange A[i+1] with A[r]   ▷ pivot goes just to the right of the low side
return i+1   ▷ new index of the pivot
```

مرتب‌سازی سریع

- زمان اجرای الگوریتم مرتب‌سازی سریع به نحوه تقسیم‌بندی آرایه بستگی دارد. اگر تقسیم‌بندی آرایه متوازن نباشد الگوریتم در زمان $\Theta(n^2)$ اجرا می‌شود اما اگر تقسیم‌بندی متوازن باشد، الگوریتم در زمان $\Theta(n \lg n)$ اجرا می‌شود.

مرتب‌سازی سریع

- اگر در هر بار تقسیم‌بندی آرایه، یک قسمت $n-1$ عنصر و قسمت دیگر 0 عنصر داشته باشد، آنگاه تقسیم‌بندی نامتوازن است. هزینه تقسیم‌بندی آرایه برابراست با $\Theta(n)$. مرتب‌سازی یک آرایه با صفر عنصر در زمان ثابت انجام می‌شود یعنی $T(0) = \Theta(1)$ بنابراین خواهیم داشت :

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

- با حل کردن این رابطه بازگشتی به دست می‌آوریم $T(n) = \Theta(n^2)$.

- بنابراین در بدترین حالت الگوریتم مرتب‌سازی سریع مانند مرتب‌سازی درجی عمل می‌کند. بدترین حالت در مرتب‌سازی سریع وقتی رخ می‌دهد که آرایه کاملا مرتب باشد.

مرتب‌سازی سریع

- اگر الگوریتم تقسیم‌بندی، آرایه را به دو قسمت تقریبا مساوی تقسیم کند، آنگاه می‌توانیم زمان اجرای الگوریتم را با استفاده از رابطه بازگشتی زیر محاسبه کنیم.

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

- با حل این رابطه به دست می‌آوریم $T(n) = \Theta(n\lg n)$.
- می‌توان اثبات کرد که الگوریتم مرتب‌سازی سریع در حالت میانگین در زمان $\Theta(n\lg n)$ اجرا می‌شود.
- همچنین برای اینکه بدترین حالت اتفاق نیافتد، می‌توان عنصر محوری را به صورت تصادفی انتخاب کرد و اثبات کرد که در این صورت زمان اجرای الگوریتم مرتب‌سازی سریع $\Theta(n\lg n)$ خواهد بود.

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 = 1 to n do   ▷ compute entries in each of n rows
2:     for j = 1 to n do     ▷ compute n entries in row i
3:         for k = 1 to n do
4:             c[i,j] = c[i,j] + a[i,k] * b[k,j]          ▷ compute c[i,j]
```

- 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 امکان 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 :

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

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.

$$\begin{aligned}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\
C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}
\end{aligned}$$

- 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.
if n==1 then
    c_11 = c_11 + a_11 * b_11
    return
  ▷ Divide.
Partition A, B, and C into n/2 × n/2 submatrices
A_11, A_12, A_21, A_22;
B_11, B_12, B_21, B_22;
C_11, C_12, C_21, C_22; respectively
```

Algorithm Matrix

```
    ▷ Conquer.
Matrix-Multiply-Recursive (A11, B11, C11, n/2)
Matrix-Multiply-Recursive (A11, B12, C12, n/2)
Matrix-Multiply-Recursive (A21, B11, C21, n/2)
Matrix-Multiply-Recursive (A21, B12, C22, n/2)
Matrix-Multiply-Recursive (A12, B21, C11, n/2)
Matrix-Multiply-Recursive (A12, B22, C12, n/2)
Matrix-Multiply-Recursive (A22, B21, C21, n/2)
Matrix-Multiply-Recursive (A22, B22, C22, n/2)
```

ضرب ماتریس‌ها

- یک ضرب ماتریسی با اندازهٔ n را به ۸ ضرب ماتریسی با اندازهٔ $n/2$ 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 $\Theta(n^{\lg 7})$ ابداع کرد. از آنجایی که $\lg 7 = 2.8073...$ ، بنابراین می‌توان گفت الگوریتم استراسن در زمان $O(n^{2.81})$ ضرب دو ماتریس را محاسبه می‌کند.

- الگوریتم استراسن یک الگوریتم از نوع تقسیم و حل است.

- استراسن مجددا در سال ۱۹۸۶ الگوریتمی از مرتبه $O(n^{2.48})$ ارائه داد. در سال ۱۹۹۰ الگوریتمی از مرتبه $O(n^{2.38})$ و در سال ۲۰۲۳ یک الگوریتم بهبود یافته ارائه شد. جستجو برای پیدا کردن سریع‌ترین الگوریتم ضرب ماتریسی همچنان ادامه دارد.

[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 دهبم. 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.

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

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.

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

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.

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.

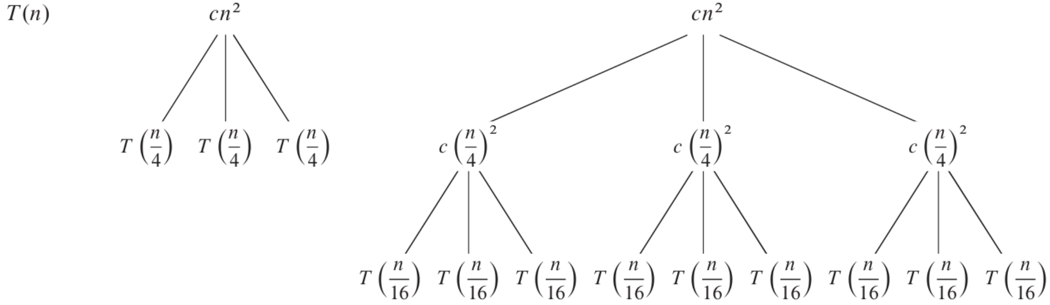
[1] recursion tree

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.

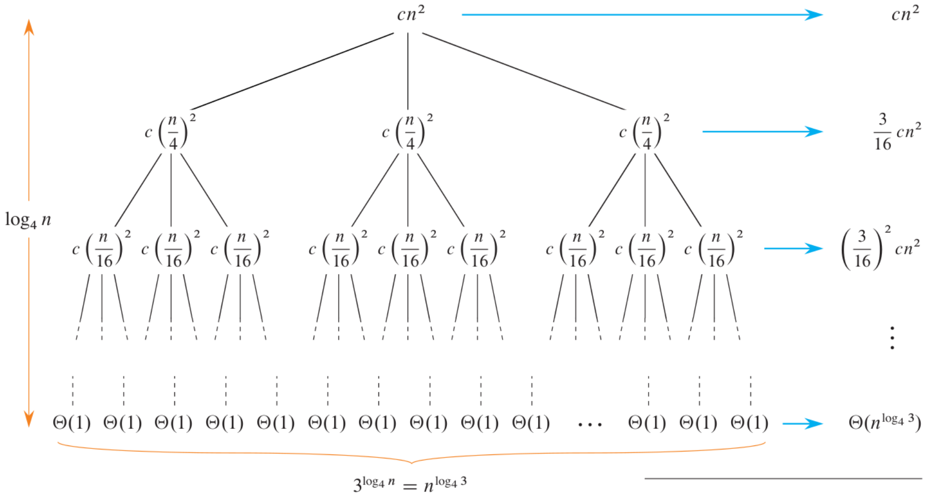
روش درخت بازگشت

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



روش درخت بازگشت

- اگر مجموع هزینه‌ها 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.

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

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

- در یک حالت خاص اگر داشته باشیم، $T(n) = aT(n/b) + cn^k$ آنگاه می‌توانیم اثبات کنیم:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & a > b^k \text{ اگر} \\ \Theta(n^k \lg n) & a = b^k \text{ اگر} \\ \Theta(n^k) & a < b^k \text{ اگر} \end{cases}$$

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 به دست می‌آوریم $n^{\log_b^a} = n^{\log_3^9} = \Theta(n^2)$. 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 $T(n) = \Theta(\lg n)$.

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 $n^{\log_2^7} = n^{\lg 7}$. از آنجایی که $\lg 7 = 2.8073...$ ، می‌توانیم قرار دهیم $\epsilon = 0.8$ و برای تابع محرک خواهیم داشت $f(n) = \Theta(n^2) = O\left(n^{\lg 7 - \epsilon}\right)$ ، پس حالت اول در قضیه اصلی برقرار است و بنابراین جواب رابطهٔ بازگشتی برابر است با $T(n) = \Theta(n^{\lg 7})$.