دانشکده‌ی علوم ریاضی

ساختمان داده‌ها ۳۱ شهریور ۹۲

جلسه‌ی ۳: روش تقسیم‌وحل

مدرّس: دکتر شهرام خزائی *نگارنده: سیاوش ریاحی و صالح سرشکی*

۱ روش تقسیم‌وحل

ساختار بسیاری از الگوریتم‌ها بازگشتی[۱] است یعنی در درون مسأله، خود را فراخوانی می‌کنند. این الگوریتم‌ها معمولاً از روش تقسیم‌وحل[۲] پیروی می‌کنند. راهکار تقسیم‌وحل، در هر سطح بازگشتی (هر مرحله‌ی بازفراخوانی الگوریتم) شامل سه مرحله‌ی تقسیم[۳] ، حل[۴] و ترکیب[۵] است:

- **تقسیم**: مسأله، به تعدادی زیرمسأله[۶] تقسیم می‌شود.
- **حل**: زیرمسأله‌ها، به صورت بازگشتی حل می‌شوند.
- **ترکیب**: جواب زیرمسأله‌ها با هم ترکیب و مسأله‌ی اولیه حل می‌شود.

۲ مرتب‌سازی ادغامی

مرتب‌سازی ادغامی[۷] از راهکار تقسیم‌وحل پیروی می‌کند. مراحل این الگوریتم به صورت زیر است:

- **تقسیم**: دنباله‌ی n عنصری که باید مرتب شود، به دو زیردنباله‌ی $n/۲$ عنصری تقسیم می‌شود.
- **حل**: دو زیردنباله، با استفاده از مرتب‌سازی ادغامی، به صورت بازگشتی مرتب می‌شوند.
- **ترکیب**: دو زیردنباله‌ی مرتب‌شده برای تولید پاسخ مرتب، ادغام می‌شوند.

زیرمسأله‌ها و فراخوانی اولیه: برای اینکه الگوریتم طراحی شده قابلیت اعمال به زیرمسأله‌ها را نیز داشته باشد، از الگوریتم $\textsc{Merge-Sort}(A,p,r)$ برای مرتب سازی زیرآرایه‌ی $A[p \cdots r]$، استفاده می‌کنیم. در اینصورت مرتب سازی آرایه‌ی ورودی $A[۱ \cdots n]$، با فراخوانی الگوریتم $\textsc{Merge-Sort}(A,۱,n)$ انجام می‌شود.

[۱] recursive
[۲] divide and conquer
[۳] divide
[۴] conquer
[۵] combine
[۶] subproblem
[۷] merge-sort

پیاده‌سازی بازگشتی $\textsc{Merge-Sort}$: الگوریتم $\textsc{Merge-Sort}(A,p,r)$، ابتدا دو زیردنباله‌ی $A[p\cdot\cdot q]$ و $A[q+۱\cdot\cdot r]$، که $q=\lfloor(p+r)/۲\rfloor$، را با فراخوانی $\textsc{Merge-Sort}(A,p,q)$ و $\textsc{Merge-Sort}(A,q+۱,r)$ به طور جداگانه مرتب می‌کند. سپس این دو زیردنباله‌ی مرتب‌شده را با فراخوانی $\textsc{Merge}(A,p,q,r)$ ترکیب و مرتب می‌کند. جزئیات الگوریتم $\textsc{Merge-Sort}(A,p,r)$ به صورت زیر است:

Algorithm 1 Merge-Sort

```
function Merge-Sort(A, p, r)
    if p < r then
        q ← ⌊(p + r)/2⌋
        Merge-Sort(A, p, q)
        Merge-Sort(A, q + 1, r)
        Merge(A, p, q, r)
```

اگر $r\le p$ الگوریتم هیچ کاری انجام نمی‌دهد، زیرا $r\le p$ معادل تهی یا تک عضوی بودن آرایه است. یک آرایه‌ی تک عضوی، به‌وضوح مرتب است و آرایه‌ی تهی (به انتفاع مقدم) مرتب است.
درصورتی که طول آرایه بیش‌تر از یک باشد، یعنی $r>p$ ابتدا کف $(p+r)/۲$ محاسبه شده و سپس دو بار، خود الگوریتم فراخوانی می‌شود. درنهایت، الگوریتم دیگری به نام فراخوانی می‌شود که دو زیرآرایه‌ی مرتب $A[p\cdot\cdot q]$ و $A[q+۱\cdot\cdot r]$ را ادغام می‌کند.
الگوریتم (A,p,q,r) به صورت زیر است:

Algorithm 2 Merge

```
1:  function Merge(A, p, q, r)
2:      [ assumes A[p..q] and A[q + 1..r] are Sorted]
3:      n1 ← q − p + 1
4:      n2 ← r − q
5:      let L[1 · ·n1 + 1] and R[1 · ·n2 + 1] be new arrays
6:      for i = 1  to n1 do
7:          L[i] ← A[p + i − 1]
8:      for j = 1  to n2 do
9:          R[j] ← A[q + j]
10:     L[n1 + 1] ← ∞
11:     R[n2 + 1] ← ∞
12:     i ← 1
13:     j ← 1
14:     for k = p  to r do
15:         if L[i] ≤ R[j] then
16:             A[k] ← L[i]
17:             i ← i + 1
18:         else
19:             A[k] ← R[i]
20:             j ← j + 1
```

در الگوریتم ادغام، ابتدا طول دو زیرآرایه برای مرتب شدن حساب می‌شود (خطوط ۲ و ۳). سپس آرایه‌های L و R (به ترتیب چپ و راست) به طول‌های $n_۱+۱$ و $n_۲+۱$ تعریف می‌شوند. آن‌گاه مقادیر لازم از آرایه‌ی A به این آرایه‌ها ریخته‌شده و در انتها دو مقدار ∞ در آن‌ها قرار می‌گیرد (اگر این دو مقدار در آرایه‌ها قرار نگیرد، در هر بار تکرار حلقه باید بررسی کنیم که آیا تمام عناصر آرایه استفاده‌شده‌است یا خیر. ولی با قرار دادن این دو مقدار، بلافاصله پس از

خواندن این مقدار، باقی مقادیر آرایه‌ی دیگر به ترتیب در ادامه‌ی A قرار می‌گیرند). در خطوط ۱۲ و ۱۳، متغیرهای i و j با مقداردهی می‌شوند تا به‌ترتیب به اولین عنصر آرایه‌های L و R اشاره کنند. در بدنه آخرین حلقه **for** (خطوط ۱۵ تا ۲۰)، هر بار مقادیر $L[i]$ و $R[j]$ با هم مقایسه شده و هرکدام کوچک‌تر بود، به آرایه A انتقال می‌یابد. بسته به اینکه کدام مقدار انتقال یافته است، یکی از اشاره‌گرهای i یا j یک واحد افزایش می‌یابد.

مثال ۱ *در زیر الگوریتم* Merge-Sort *روی آرایه‌ی* [۹, ۴, ۱۲, ۶, ۳, ۱۴, ۱۰, ۱۳] *اجرا شده است و تغییرات مرحله‌به‌مرحله روی آرایه به ترتیب فراخوانی* Merge *نشان داده شده است.*

initial array	9	4	12	6	3	14	10	13
Merge$(A,1,1,2)$	***4***	***9***	12	6	3	14	10	13
Merge$(A,3,3,4)$	4	9	***6***	***12***	3	14	10	13
Merge$(A,1,2,4)$	***4***	***6***	***9***	***12***	3	14	10	13
Merge$(A,5,5,6)$	4	6	9	12	***3***	***14***	10	13
Merge$(A,7,7,8)$	4	6	9	12	3	14	***10***	***13***
Merge$(A,5,6,8)$	4	6	9	12	***3***	***10***	***13***	***14***
Merge$(A,1,4,8)$	***3***	***4***	***6***	***9***	***10***	***12***	***13***	***14***

۳ اثبات درست‌بودن Merge

برای اثبات درستی الگوریتم، از ناوردایی (ثابت حلقه) زیر استفاده می‌کنیم.
ناوردایی: درست پس از مقداردهی متغیر آخرین حلقه **for** با مقدار k، در خط ۱۴، زیرآرایه‌ی $A[p\cdots k-1]$ شامل $k-p$ کوچک‌ترین عنصر آرایه‌های $L[1\cdots n_1+1]$ و $R[1\cdots n_2+1]$ به صورت مرتب است. علاوه براین، $L[i]$ و $R[j]$ کوچک‌ترین عناصر آرایه‌های خودشان هستند که در A کپی نشده‌اند.
باید نشان دهیم که این ثابت حلقه، قبل از اولین تکرار حلقه‌ی **for** در خط ۱۴ برقرار است، هر تکرار حلقه این ثابت حلقه را حفظ می‌کند و پس از پایان حلقه نیز این ثابت حفظ می‌شود.

- **آغاز:** قبل از اولین تکرار حلقه داریم $k=p$، به‌طوری که زیرآرایه‌ی $A[p\cdots k-1]$ خالی است. این زیرآرایه‌ی خالی، شامل $k-p=0$ کوچک‌ترین عنصر آرایه‌های L و R است. چون $i=j=1$، هم $L[i]$ و هم $R[j]$ کوچک‌ترین عناصر آرایه‌های خودشان هستند که هنوز در A کپی نشده‌اند.

- **نگه‌داری:** باید نشان دهیم تکرار حلقه، ثابت حلقه (ناوردایی) را حفظ می‌کند. یعنی باید نشان دهیم که اگر در آغاز حلقه با مقادیر i، j و k ناوردایی برقرار باشد، پس از پایان حلقه (آغاز حلقه‌ی بعدی) نیز ناوردایی برای مقادیر جدید i، j و k برقرار است.

  برای اثبات، دو حالت $L[i]\le R[j]$ و $L[i]>R[j]$ را جداگانه بررسی می‌کنیم.

  اگر $L[i]\le R[j]$، آن‌گاه $L[i]$ کوچک‌ترین عنصری است که هنوز در A کپی نشده‌است. حال چون در خط ۱۶ $L[i]$ در $A[k]$ کپی شده‌است و $A[p\cdots k-1]$ شامل $k-p$ کوچک‌ترین عنصر است، زیرآرایه‌ی $A[p\cdots k]$ شامل $k+1-p$ کوچک‌ترین عنصر خواهدبود.

  برای حالت $L[i]>R[j]$، به‌طور مشابه ثابت می‌شود.

- **خاتمه:** در پایان $k=r+1$، بنابراین داریم $A[p\cdots k-1]=A[p\cdots r]$. بنابر ثابت حلقه، آرایه‌ی $A[p\cdots r]$ شامل $k-p=r-p+1$ کوچک‌ترین عنصر $L[1\cdots n_1+1]$ و $R[1\cdots n_2+1]$ است، پس مرتب است.

۴ پیاده‌سازی غیر بازگشتی Merge-Sort

پیاده‌سازی بازگشتی الگوریتم مرتب‌سازی ادغامی اصطلاحاً بالابه‌پایین[۸] نامیده می‌شود. ترتیب فراخوانی‌های الگوریتم Merge در مثال ۱ خود گویای علت این نامگذاری است. الگوریتم مرتب‌سازی ادغامی را می‌توان به صورت غیر بازگشتی[۹] نیز پیاده‌سازی کرد. برای این کار باید ترتیب فراخوانی‌های Merge را اصطلاحاً به صورت پایین‌به‌بالا[۱۰] انجام داد. برای این کار، ابتدا باید آرایه‌ی ورودی را به زیرآرایه‌های دو عضوی تقسیم کنیم و زیر آرایه‌های دو عضوی را مرتب کنیم. سپس، زیرآرایه‌های دو عضوی را (با استفاده از تابع Merge) ادغام می‌کنیم تا زیرآرایه‌های ۴ عضوی، ایجاد شود و زیرآرایه‌های ۴ عضوی ایجاد شده را مرتب و ادغام می‌کنیم. سپس، این روند را برای زیرآرایه‌های ۸ عضوی و بالاتر ادامه می‌دهیم، تا آرایه‌ی اصلی مرتب شود. الگوریتم زیر جزئیات این نحوه پیاده‌سازی را نشان می‌دهد.

Algorithm 3 Bottom-Up-Merge-Sort

function Bottom-Up-Merge-Sort(A, n)
    $d \leftarrow 1$
    while $(d < n)$ **do**
        for $i = 1$ **to** $n - d$ **with steps** $2d$ **do**
            $p \leftarrow \min(i + d - 1, n)$
            $q \leftarrow \min(i + 2d - 1, n)$
            Merge(A, i, p, q)
        $d \leftarrow 2d$

مثال ۲ *در زیر الگوریتم غیربازگشتی* Merge-Sort *روی آرایه‌ی* [۹, ۴, ۱۲, ۶, ۳, ۱۴, ۱۰, ۱۳] *اجرا شده است و تغییرات مرحله‌به‌مرحله روی آرایه به ترتیب فراخوانی* Merge *نشان داده شده است.*

	initial array				9,	4,	12,	6,	3,	14,	10,	13
$d = 1,$	$i = 1,$	$p = 1,$	$q = 2$	Merge$(A, 1, 1, 2)$	***4***,	***9***,	12,	6,	3,	14,	10,	13
$d = 1,$	$i = 3,$	$p = 3,$	$q = 4$	Merge$(A, 3, 3, 4)$	4,	9,	***6***,	***12***,	3,	14,	10,	13
$d = 1,$	$i = 5,$	$p = 5,$	$q = 6$	Merge$(A, 5, 5, 6)$	4,	9,	6,	12,	***3***,	***14***,	10,	13
$d = 1,$	$i = 7,$	$p = 7,$	$q = 8$	Merge$(A, 7, 7, 8)$	4,	9,	6,	12,	3,	14,	***10***,	***13***
$d = 2,$	$i = 1,$	$p = 2,$	$q = 4$	Merge$(A, 1, 2, 4)$	***4***,	***6***,	***9***,	***12***,	3,	14,	10,	13
$d = 2,$	$i = 5,$	$p = 6,$	$q = 8$	Merge$(A, 5, 6, 8)$	4,	6,	9,	12,	***3***,	***10***,	***13***,	***14***
$d = 4,$	$i = 1,$	$p = 4,$	$q = 8$	Merge$(A, 1, 4, 8)$	***3***,	***4***,	***6***,	***9***,	***10***,	***12***,	***13***,	***14***

همان‌طور که مشاهده می‌شود، ترتیب اجرای Merge-Sort در حالت بازگشتی و غیر بازگشتی متفاوت است، به گونه‌ای که حالت غیر بازگشتی، شامل ادغام جفت‌هایی از آرایه‌های یک عنصری برای ایجاد آرایه‌های مرتب دو عنصری، ادغام آرایه‌های دو عنصری برای ایجاد آرایه‌های چهار عنصری است و درنهایت دو آرایه‌ی مرتب چهار عنصری، با هم ادغام و آرایه‌ی مرتب نهایی ایجاد می‌شود. در حالت بازگشتی، ابتدا کل آرایه به دو زیرآرایه تقسیم شده و هرکدام جداگانه مرتب می‌شوند. سپس این دو زیرآرایه‌ی مرتب‌شده، با هم ادغام می‌شوند.

مقایسه پیاده‌سازی بالابه‌پایین و پایین‌به‌بالا . پیاده‌سازی بالابه‌پایین، به دلیل استفاده از فراخوانی‌های بازگشتی نیاز با حافظه بیشتری دارد. اما از نظر کارایی پیاده‌سازی پایین‌به‌بالا اندکی از پیاده‌سازی بالابه‌پایین در عمل کندتر است. جدول زیر این مقایسه را روی یک پردازنده نوعی بر حسب ثانیه نشان می‌دهد.

[۸] top-down
[۹] non-recursive
[۱۰] bottom-up

n	top-down	bottom-up
۱۰۰٬۰۰۰	۵۳	۵۹
۲۰۰٬۰۰۰	۱۱۱	۱۲۷
۴۰۰٬۰۰۰	۲۳۷	۲۶۷
۸۰۰٬۰۰۰	۵۲۴	۵۶۸

۵ محاسبه‌ی پیچیدگی الگوریتم Merge-Sort

برای محاسبه‌ی پیچیدگی الگوریتم‌های بازگشتی، باید رابطه‌ی بازگشتی[۱۱] مناسب برای الگوریتم را حل کنیم. در روش تقسیم و حل معمولاً به رابطه‌ی زیر می‌رسیم:

$$T(n) = aT(n/b) + D(n) + C(n)$$
$$T(۱) = c$$

در این رابطه، $D(n)$ زمان اجرای مرحله‌ی تقسیم است، $C(n)$ زمان اجرای مرحله‌ی ترکیب است و ضریب a تعداد زیرمسأله‌ها است که اندازه‌ی هر کدام، $۱/b$ اندازه‌ی مسأله‌ی اصلی است.
در Merge-Sort داریم:

- **تقسیم**: در مرحله‌ی تقسیم، فقط وسط آرایه محاسبه می‌شود، پس در زمان ثابتی اجرا می‌شود. بنابراین داریم: $D(n) = \Theta(۱)$.
- **حل**: در هر بار فراخوانی Merge-Sort آرایه‌ی مورد نظر، به دو زیرآرایه با اندازه‌ی $n/۲$ تقسیم شده و برای هرکدام، Merge-Sort فراخوانی می‌شود. پس دو زیرمسأله ($a = ۲$) با اندازه‌ی $n/۲$ ($b = ۲$)خواهیم داشت.
- **ترکیب**: در هر بار اجرای Merge-Sort برای ترکیب زیرمسأله‌ها، الگوریتم Merge فراخوانی می‌شود. می‌دانیم که این الگوریتم در زمان $\Theta(n)$ اجرا می‌شود. بنابراین داریم: $C(n) = \Theta(n)$.

پس، رابطه‌ی بازگشتی Merge-Sort به این صورت می‌شود:

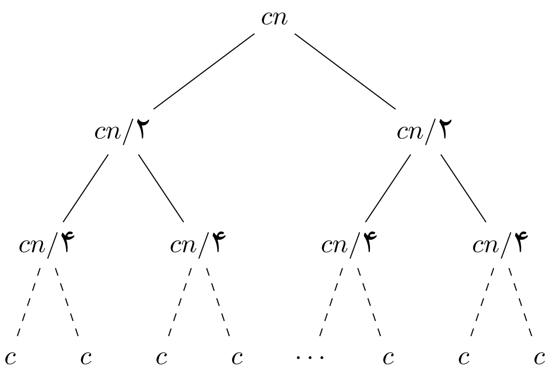
$$T(n) = \begin{cases} ۲T(n/۲) + \Theta(n) + \Theta(۱) & n > ۱ \\ \Theta(۱) & n = ۱ \end{cases}$$

ولی در عمل تابع $\Theta(۱)$ ، جذب $\Theta(n)$ می‌شود. با تعریف Θ در جلسات بعد بیش‌تر آشنا می‌شویم و به علت این جذب، پی می‌بریم. بنابراین خواهیم داشت:

$$T(n) = \begin{cases} ۲T(n/۲) + \Theta(n) & n > ۱ \\ \Theta(۱) & n = ۱ \end{cases}$$

برای حل این عبارت بازگشتی از درخت بازگشتی[۱۲] استفاده می‌کنیم که حجم کل محاسبات را در مراحل مختلف نشان می‌دهد. برای سادگی فرض کنید که تابع cn تابعی است که به‌جای $\Theta(n)$ در رابطه فوق قرار می‌گیرد. یعنی ترکیب جواب‌های زیر مسأله‌های با اندازه $n/۲$ در زمان cn انجام می‌شود. همچنین فرض کنید c مقدار ثابتی باشد که به‌جای $\Theta(۱)$ قرار می‌گیرد. یعنی حل کردن زیر مسأله‌های با اندازه ۱ در زمان c انجام می‌شود. در اینصورت درخت بازگشتی به شکل زیر است:

[۱۱] recurrence relation
[۱۲] recursion-tree



با توجه به رابطه‌ی بازگشتی Merge-Sort می‌دانیم که در هر بار اجرای این الگوریتم، دوبار الگوریتم Merge-Sort فراخوانی شده، ولی اندازه‌ی آرایه‌ی هر کدام، نصف می‌شود. پس در فراخوانی jام، اندازه‌ی هر زیرمسأله، برابر $\frac{n}{۲^j}$ و تعداد زیرمسأله‌ها، برابر ۲^j می‌شود. بنابراین، کل هزینه‌ی اجرای مرحله‌ی jام، برابر است با:

$$۲^j c \frac{n}{۲^j} = cn$$

از طرفی تعداد مراحل الگوریتم، یک واحد بیش‌تر از ارتفاع درخت بازگشتی است. برای آن‌که محاسبات ساده‌تر شود، فرض می‌کنیم n دقیقاً توانی از ۲ است ($n = ۲^h$)، که h ارتفاع درخت بازگشتی است. بنابراین تعداد سطوح درخت برابر $h + ۱ = ۱ + \log n$ است.
درنتیجه کل هزینه‌ی اجرای Merge-Sort برابر است با:

$$cn(۱ + \log n) = cn \log n + cn = \Theta(n \log n)$$

در جلسات بعد پس از آشنایی با روش اصلی[۱۳] می‌توان رابطه‌ی بازگشتی الگوریتم Merge-Sort را به طور دقیق و ساده حل کرد، که با این روش نیز، جواب برابر عبارت فوق می‌شود.

[۱۳] master method