دانشکده‌ی علوم ریاضی

ساختمان داده	۱۴ آبان ۱۳۹۲

جلسه‌ی ۱۳ب: مرتب‌سازی شمارشی و مبنایی

مدرّس: دکتر شهرام خزائی *نگارنده: جواد عابدی گزل‌آباد*

۱ مقدمه

در این جلسه قصد داریم که با دو الگوریتم مرتب‌سازی دیگر آشنا شویم. تفاوت این دو الگوریتم با الگوریتم‌های قبلی محدودیت‌هایی است که در اعداد ورودی وجود دارد. در الگوریتم‌های مقایسه‌ای محدودیتی روی اعداد نداشتیم و به همین دلیل برای مرتب‌سازی نیازمند مقایسه اعداد با یکدیگر بودیم. در حالی که در بسیاری از مسائل، اعداد ورودی در بازه‌ی مشخصی قرار دارند و در نتیجه برای مرتب‌سازی آنها می‌توان روش‌های بهینه‌تری نسبت به قبل ارائه کرد.

در این جلسه توجه خود را به الگوریتم‌های مرتب‌سازی معطوف می‌کنیم که فرض می‌کنند اعداد ورودی در بازه‌ی $[0,k]$ قرار دارند. ابتدا با مرتب‌سازی شمارشی[۱] آشنا می‌شویم که الگوریتمی با پیچیدگی $\Theta(n+k)$ ارائه می‌کند. این الگوریتم برای مقادیر $k=O(n)$ خطی است، اما برای مقادیر $k=O(n^c)$ (که c ثابت دلخواه است) خطی نیست. در ادامه مرتب‌سازی مبنایی[۲] را مورد بررسی قرار می‌دهیم که برای مقادیر $k=O(n^c)$ نیز خطی است.

۲ مرتب‌سازی شمارشی

مرتب‌سازی شمارشی یکی از الگوریتم‌های مرتب‌سازی است که با فرض دانستن بازه کلیدهای آرایه‌ی ورودی (آرایه‌ی A)، عمل مرتب‌سازی را انجام می‌دهد. به طور دقیق‌تر مساله‌ی زیر را در نظر بگیرید:

[۱] Counting Sort
[۲] Radix Sort

- **ورودی:** دنباله‌ی $a_1, a_2, ..., a_n$ از اعداد صحیح و عدد صحیح k که $a_i \in [0, k]$.
- **خروجی:** جایگشت مرتب‌شده‌ی $b_1, b_2, ..., b_n$ از ورودی.

این الگوریتم از بازه‌ی موجود برای ساختن یک آرایه به نام C استفاده می‌کند (اندازه‌ی آرایه‌ی C برابر با طول بازه است). عضو iام در آرایه C برابر تعداد کلیدهای A با مقدار i است. در نهایت با استفاده از آرایه‌ی C مرتب‌سازی انجام می‌شود.

توجه کنید که مرتب‌سازی شمارشی از مقادیر به عنوان اندیس آرایه استفاده می‌کند. در نتیجه یک الگوریتم مقایسه‌ای نیست و بنابراین کران پایین $\Omega(n \log n)$ برای این الگوریتم قابل تطبیق نیست.

۱.۲ الگوریتم

1. آرایه‌ی C را به اندازه‌ی k ساخته و با 0 مقداردهی اولیه کن.
2. به ازای هر عدد در آرایه‌ی A همانند a_i به خانه‌ی $C[a_i]$ یک واحد اضافه کن.
3. به انتهای آرایه‌ی خروجی به ازای عنصر iام، $C[i]$ تا عدد i اضافه کن.

```
function CountingSort(A[1...n], k)
    for j = 0 to k do
        C[j] ← 0
    for j = 1 to n do
        C[A[j]] ← C[A[j]] + 1
    count ← 1
    for j = 0 to k do
        for i = 1 to C[j] do
            B[count] ← j
            count ← count + 1
    return B
```

پس از اعمال الگوریتم آرایه‌های B و C مقداردهی خواهند شد:

- آرایه‌ی C: آرایه‌ای به طول $k + 1$ است که مقدار خانه‌ی iام آن برابر با تعداد عناصر موجود در آرایه‌ی A با کلید i است.
- آرایه‌ی B: آرایه‌ی خروجی که باید عناصر آرایه‌ی A را به صورت مرتب‌شده در خود ذخیره کند.

۲.۲ ویژگی‌های الگوریتم

در مرتب‌سازی شمارشی از دو حلقه به اندازه‌ی k و n استفاده می‌شود. همچنین در حلقه‌ی سوم حداکثر $n+k$ عملیات انجام می‌شود. در نتیجه این الگوریتم از $\Theta(n+k)$ است.

نکته ۱ *در صورتی که k خیلی بزرگتر از n باشد این روش مرتب‌سازی بهینه نیست ولی در حالتی که $k = O(n)$ باشد، این الگوریتم از $O(n)$ خواهد بود و یک الگوریتم خطی خواهیم داشت که بهینه‌ترین روش مرتب‌سازی بدست آمده است.*

با توجه به اینکه آرایه‌ی C تعداد وقوع هر عدد در آرایه‌ی A را بیان می‌کند، عملا استفاده از مرتب‌سازی شمارشی برای بازه‌های بزرگ اعداد غیرعملی می‌شود. به عنوان یک مثال، مرتب‌سازی شمارشی می‌تواند بهترین الگوریتم برای اعدادی باشد که بین ۰ تا ۱۰۰ قرار دارند. این الگوریتم برای مرتب کردن اسامی براساس حروف الفبا نامناسب است.

این الگوریتم، یک الگوریتم مرتب‌سازی پایدار است. به این معنی که ترتیب عناصر با کلید یکسان را حفظ می‌کند. چرا که در هنگام ذخیره‌سازی در آرایه‌ی C ترتیب نگهداری آنها تغییری نمی‌کند و در هنگام ذخیره‌سازی اطلاعات در آرایه‌ی B نیز همان ترتیب رعایت می‌شود. با این وجود الگوریتم فوق برای حالتی که ورودی آرایه‌ی از رکوردها باشد و مرتب‌سازی بخواهد از روی یکی از مولفه‌های آن انجام شود، قابل تطبیق نمی‌باشد.

برای رفع این مشکل روش دیگری برای پیاده‌سازی این الگوریتم وجود دارد. در این روش آرایه‌ی C را به گونه‌ی دیگری تعریف می‌کنیم:

$C[i]$: تعداد عناصر موجود در آرایه‌ی A با کلید کمتر (یا مساوی) i است. در واقع آرایه‌ی C اندیس آخرین خانه از آرایه‌ی B با مقدار i را نگهداری می‌کند.

بدین ترتیب آرایه‌ی C فرکانس تجمعی آرایه‌ی ورودی را محاسبه می‌کند. با این تغییرات در الگوریتم پیاده‌سازی زیر را داریم:

function $\textsc{CountingSort}(A, k)$
  for $j = ۰$ **to** k **do**
    $C[j] \leftarrow ۰$
  for $j = ۱$ **to** n **do**
    $C[A[j]] \leftarrow C[A[j]] + ۱$
  for $j = ۰$ **to** k **do**
    $C[j] \leftarrow C[j] + C[j-۱]$
  for $j = n$ **downto** ۱ **do**
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - ۱$
  return B

با تغییر صورت گرفته این الگوریتم به راحتی قابل تطبیق برای حالتی است که ورودی آرایه‌ای رکوردها باشد.
تعداد عملیاتی که این روش مرتب‌سازی انجام می‌دهد تفاوتی با قبل ندارد و از همان مرتبه است.

در این حالت، در انتها مقادیر آرایه‌ی C صفر نخواهند شد. بلکه به ازای هر iای، $C[i]$ برابر است با اندیس اولین خانه از آرایه‌ی B با مقدار i.

۳ مرتب‌سازی مبنایی

در این الگوریتم نیز همانند الگوریتم قبلی محدودیتی روی اعداد ورودی وجود دارد و با فرض این محدودیت، مسئله را حل می‌کنیم. در این مرتب‌سازی ورودی‌ها را صورت اعداد d رقمی در نظر می‌گیریم. سپس اعداد را ابتدا براساس کم‌ارزش‌ترین رقم مرتب می‌کنیم، سپس براساس دومین رقم تا در نهایت براساس پرارزش‌ترین رقم آن. در اینصورت اگر از یک مرتب‌سازی پایدار استفاده شود، می‌توان نشان داد که اعداد نهایتاً در d مرحله مرتب می‌شوند.
این روش مرتب‌سازی، خود نیز پایدار است و در تهیه‌ی واژه‌نامه‌ها و مرتب‌سازی اعداد بسیار کاربرد دارد. هرمان هولریث[۳] اولین فردی بود که در سال ۱۸۸۷ روی ماشین‌های جدول‌بندی از این الگوریتم استفاده کرد.

۱.۳ الگوریتم

به صورت کلی مرتب‌سازی مبنایی به صورت زیر عمل می‌کند که در آن A یک لیست n تایی از اعداد d رقمی در مبنای b است (در واقع هر عدد بین ۰ تا $b^d - ۱$ است).

function RadixSort(A, d, b)
  for $j = ۱$ **to** d **do**
    Sort A using a stable sort on ith digit

مثال ۱ *فرض کنید هفت عدد سه رقمی داریم که می‌خواهیم آنها را مرتب کنیم:*

[۳] Herman Hollerith

Radix Sort			
init	*1st*	*2nd*	*final*
329	*720*	*720*	*329*
457	*355*	*329*	*355*
657	*436*	*436*	*436*
839	*457*	*839*	*457*
436	*657*	*355*	*657*
720	*329*	*457*	*720*
355	*839*	*657*	*839*

در این مثال در هر مرحله اعداد در یک ستون قرار دارند. ستون سمت چپ حالت ابتدایی است. در مرحله اول اعداد برحسب رقم یکان، سپس براساس رقم دهگان و نهایتاً تمامی اعداد براساس رقم صدگان مرتب شده‌اند.

۲.۳ تحلیل درستی الگوریتم

اولین سوالی که مطرح می‌شود این است که چرا این الگوریتم، آرایه‌ی ورودی را مرتب می‌کند. به کمک استقرا ثابت می‌کنیم که پس از مرحله‌ی jام آرایه‌ی A برحسب j رقم اول خود مرتب شده است.

- **پایه:** به ازای $j = ۱$ اعداد براساس رقم یکان مرتب می‌شوند که همان رقم اول اعداد است.
- **گام:** فرض کنید مرحله‌ی jام انجام شده است و اعداد براساس j رقم اول خود مرتب هستند. ثابت می‌کنیم پس از گام $j + ۱$ام اعداد براساس $j + ۱$ رقم اول خود مرتب خواهند شد.

در این مرحله اعداد برحسب رقم $j + ۱$ ام خود مرتب می‌شوند. در نتیجه هر دو عددی که رقم $j + ۱$شان متفاوت باشد نسبت به هم مرتب هستند. تنها باید اعدادی را بررسی کنیم که $j + ۱$امین رقمشان برابر است. در چنین حالتی باید دو عدد براساس رقم‌های قبلی خود مرتب بشوند. از طرفی طبق فرض استقرا این دو عدد نسبت به هم مرتب هستند و می‌دانیم که مرتب‌سازی در این مرحله یک مرتب‌سازی پایدار است پس موقعیت این اعداد نسبت به هم تغییری نخواهد کرد. در نتیجه اثبات شد که تمامی اعداد براساس $j + ۱$ رقم اول خود مرتب شدند.

۳.۳ تحلیل زمان اجرای الگوریتم

در هر مرحله از یک مرتب‌سازی پایدار استفاده می‌کنیم. این الگوریتم باید آرایه‌ای از اعداد را مرتب کند که هر عدد بین ۰ و $b - ۱$ است. بهترین الگوریتم با این شرایط، مرتب‌سازی شمارشی است که

از $\Theta(n+b)$ عملیات خواهد بود. در نتیجه پیچیدگی الگوریتم اصلی برابر با $\Theta(d(n+b))$ می‌شود.

۴.۳ تعمیم مرتب‌سازی مبنایی

فرض کنید که تمامی اعداد ورودی در مبنای دو نوشته شده‌اند ($b = ۲$). طبق فرمول بدست آمده زمان اجرای الگوریتم برابر $\Theta(d(n+۲)) = \Theta(dn)$ خواهد بود. برای اینکه روند اجرای الگوریتم را سریع‌تر کنیم تغییری در اعداد ایجاد می‌کنیم بدین ترتیب که مبنای اعداد موجود را از ۲ به ۲^r تغییر می‌دهیم. در واقع هر r بیت یک رقم جدید می‌شود.

از آنجایی که تعداد ارقام در مبنای جدید $\lceil \frac{d}{r} \rceil$ است، پیچیدگی الگوریتم برابر است با:

$$\Theta(\lceil \tfrac{d}{r} \rceil (n + ۲^r))$$

که به ازای $r = \log n$ بهینه با زمان اجرای $\Theta(\frac{dn}{\log n})$ خواهد بود. زیرا اگر $r < \log n$ باشد، $\lceil \frac{d}{r} \rceil (n+۲^r) \geq \frac{dn}{\log n}$ است؛ اگر مثلاً $r > ۲ \log n$ باشد، $\lceil \frac{d}{r} \rceil (n+۲^r) = \Omega(n^۲)$.

برای اینکه متغیر d را نیز از تحلیل خارج کنیم، فرض کنید که بازه‌ی اعداد ورودی از $O(n^c)$ هستند که c یک عدد ثابت است. در نتیجه $d = O(c \log n)$. با این حساب پیچیدگی الگوریتم به $O(cn) = O(n)$ تغییر پیدا می‌کند که بر حسب ورودی خطی است. به صورت خلاصه ما توانستیم n عدد که همگی کمتر از n^c هستند را در زمان اجرای خطی مرتب کنیم.