دانشکده‌ی علوم ریاضی

داده‌ساختارها و الگوریتم‌ها — ۳ آبان ۹۳

جلسه‌ی ۹: الگوریتم درجا

مدرّس: دکتر شهرام خزائی — نگارنده: سها سادات مهدی و دنیا حمزئیان

مقدمه

در علوم کامپیوتر الگوریتم درجا به الگوریتمی گفته می‌شود که برای تبدیل ورودی به خروجی، از داده ساختاری استفاده می‌کند که به مقدار کوچك و ثابتی از حافظه‌ای اضافه بر حافظه‌ی ورودی نیاز داشته باشد. معمولاً الگوریتم طوری طراحی شده‌است که در حین اجرای آن، خروجی مطلوب، از بازنویسی حافظه‌ی اختصاص یافته به ورودی به‌دست بیاید و در نتیجه، حافظه‌ی جدیدی که متغیر با اندازه‌ی ورودی باشد به آن‌ها اختصاص داده نمی‌شود.
از نکات مثبت الگوریتم‌های درجا می‌توان اشغال کردن حافظه‌ی کمتر را که ثابت و مستقل از اندازه‌ی ورودی است، نام برد. این نوع الگوریتم برای دستگاه‌هایی که حافظه‌ی محدود دارند بسیار مناسب است. از طرف دیگر، گاهی اوقات زمان اجرای الگوریتم درجا بیشتر از الگوریتم های غیردرجا است. باید با توجه به ویژگی‌های دستگاه مورد استفاده و خواسته‌های مسئله الگوریتم مناسب را انتخاب نمود.
حال با ذکر یك مثال ساده به بررسی الگوریتم‌های درجا و غیردرجا می‌پردازیم. مسئله‌ی معکوس‌سازی[1] زیر را در نظر بگیرید:

- **ورودی:** آرایه‌ی A به طول n
- **خروجی:** آرایه‌ی B که معکوس آرایه‌ی A است، به طوری که $\forall 1 \leq k \leq n : B[k] = A[n+1-k]$

در ابتدا الگوریتم غیردرجا را برای حل مطرح می‌کنیم به این صورت که یك آرایه‌ی جدید به طول n ایجاد می‌کنیم و عنصر i ام آن رابرابر عنصر $n+1-i$ ام آرایه‌ی ورودی قرار می‌دهیم.

Algorithm 1 NOT-INPLACE-REVERSE

```
function Reverse(A[1...n])
    for i = 1 to n do
        B[i] ← A[n + 1 − i]
```

این الگوریتم به حافظه‌ی اضافی از مرتبه‌ی n نیاز دارد. حال به دنبال الگوریتمی می‌گردیم که حافظه‌ی کمتر و ثابتی را اشغال کند. در این الگوریتم عنصر i ام و $n+1-i$ ام ورودی را با یکدیگر جابه‌جا می‌کنیم. برای این کار به یك واحد حافظه‌ی اضافه نیاز داریم تا به عنوان واسط جابه‌جایی عناصر از آن استفاده کنیم.

- **ورودی:** آرایه‌ی A به طول n
- **خروجی:** معکوس آرایه‌ی ورودی که روی A بازنویسی می‌شود.

Algorithm 2 INPLACE-REVERSE

```
function Reverse(A[1...n])
    for i = 1 to ⌊n/2⌋ do
        A[i] ⟷ A[n + 1 − i]
```

[1] Reverse

همان طور که می‌بینیم الگوریتم درجا تنها به یك واحد حافظه‌ی اضافه بر حافظه‌ی اختصاص‌یافته به ورودی نیاز دارد، زیرا بدنه‌ی حلقه درواقع به صورت زیر پیاده‌سازی می‌شود.

$$tmp \leftarrow A[n + ۱ - i]$$
$$A[n + ۱ - i] \leftarrow A[i]$$
$$A[i] \leftarrow tmp$$

پس حافظه‌ی اشغال‌شده از مرتبه‌ی یك است.

۱ مثال‌ها

الگوریتم‌های مرتب‌سازی انتخابی [۲] و مرتب‌سازی درجی [۳] از نمونه‌های الگوریتم‌های درجا هستند که در ادامه مختصرا شرح داده می‌شوند.

۱.۱ مرتب‌سازی انتخابی

- **ورودی:** آرایه‌ی A به طول n
- **خروجی:** آرایه‌ی مرتب‌شده‌ی صعودی A

الگوریتم مرتب‌سازی انتخابی، آرایه‌ی ورودی را به دو بخش تقسیم می‌کند؛ زیرآرایه‌ای که صعودی مرتب شده است، و زیر آرایه‌ای که بقیه‌ی عناصر پردازش‌نشده را دارد. در ابتدا بخش اول، عنصری ندارد و بخش دوم کل آرایه‌ی ورودی را دربرمی‌گیرد. الگوریتم در هر مرحله کوچکترین عنصر را از بخش مرتب‌نشده پیدا می‌کند و با اولین عنصر بخش مرتب‌نشده جابه‌جا مي‌کند و مرز بخش مرتب‌شده را یك واحد اضافه مي‌کند. بنابراین از زیرآرایه‌ی مرتب‌نشده یك عنصر حذف می‌شود. بدین ترتیب در مرحله‌ی آخر، بخش مرتب نشده عنصری ندارد و همه‌ی عناصر به طور مرتب‌شده در بخش اول قرار می‌گیرند.

Algorithm 3 SELECTION-SORT

```
function SELECTION-SORT(A[1...n])
    for i = 1 to n do
        smallest ← A[i]
        for j = i + 1 → n do
            if A[j] < A[smallest] then
                smallest ← j
        A[smallest] ⟷ A[i]
```

به‌وضوح این الگوریتم به تعداد ثابتی حافظه، اضافه بر حافظه‌ی ورودی نیاز دارد؛ پس این الگوریتم به حافظه‌ای از مرتبه‌ی یك نیاز دارد و یك الگوریتم درجا است.

۲.۱ مرتب‌سازی درجی

در الگوریتم مرتب‌سازی درجی، تنها به یک واحد حافظه علاوه بر حافظه‌ی مورد نیاز برای ذخیره‌ی ورودی نیاز است که یعنی این الگوریتم درجا است. عناصر آرایه در این الگوریتم با هم جابه‌جا می‌شوند و نیازی به آرایه‌ی کمکی نیست.

- **ورودی:** آرایه‌ی A به طول n
- **خروجی:** آرایه‌ی مرتب‌شده‌ی صعودی بازنویسی‌شده در A

[۲] Selection Sort
[۳] Insertion Sort

Algorithm 4 INSERTION-SORT

```
function INSERTION-SORT(A[1...n])
    for j = 2 to n do
        key ← A[j]
        #Insert A[j] into the sorted sequence A[1...j − 1]
        i ← j − 1
        while i > 0 and A[i] > key do
            A[i + 1] ← A[i]
            i ← i − 1
        A[i + 1] ← key
```

۳.۱ تصادفی سازی

الگوریتم تصادفی‌سازی[۴] می‌تواند به دو صورت درجا و غیردرجا پیاده سازی شود. در ابتدا حالت غیردرجای آن را شرح می‌دهیم: آرایه‌ای کمکی به طول n در نظر می‌گیریم که دارای ۲ پارامتر مقدار (val) و اولویت ($priority$) است که پارامتر مقدار برابر با مقدار نظیرش در آرایه‌ی ورودی است. سپس برای هر پارامتر مقدار یک عدد تصادفی بین ۱ و $n^۲$ انتخاب می شود و آرایه‌ی کمکی را با توجه به این پارامتر مرتب می‌کنیم. این الگوریتم دارای پیچیدگی زمانی $\Theta(n)$ است و $O(n)$ واحد حافظه‌ی اضافه می‌گیرد.

- **ورودی:** آرایه‌ی A به طول n
- **خروجی:** جایگشتی تصادفی از آرایه‌ی A

Algorithm 5 NOT-INPLACE-RANDOMIZE

```
function RANDOMIZE(A[1...n])
    for i = 1 to n do
        X[i].val ← A[i]
        X[i].priority ← RANDOM(1,n^2)
        Sort X with respect to "priority"
        Output ⟨X[1].val, ..., X[n].val⟩
```

اگر بخواهیم الگوریتم تصادفی‌سازی را به صورت درجا پیاده‌سازی کنیم، به این صورت عمل می‌کنیم که هر کدام از خانه‌های آرایه‌ی ورودی را با یکی از خانه‌های بعدی آن که اندیس آن به صورت تصادفی انتخاب می‌شود، جابه‌جا می‌کنیم.

Algorithm 6 INPLACE-RANDOMIZE

```
for i = 1 to n do
    j ← RANDOM(i, n)
    A[i] ← A[j]
```

۴.۱ مرتب‌سازی ادغامی

در مرتب‌سازی ادغامی[۵] به یک آرایه‌ی کمکی جهت ادغام دو آرایه‌ی مرتب‌شده نیاز داریم؛ پس این الگوریتم درجا نیست. برای درجا شدن این الگوریتم، تابع IN-PLACE-MERGE را تعریف می‌کنیم.

- **ورودی:** آرایه‌ی A به طول n به طوری که نیمه‌ی سمت راست و چپ آن دو زیرآرایه‌ی مرتب‌شده هستند.
- **خروجی:** آرایه‌ی مرتب‌شده‌ی صعودی A

[۴] Randomize
[۵] Merge Sort

در این تابع، ادغام به صورت درجا صورت می‌گیرد؛ یعنی ابتدا مرز بین دو زیرآرایه‌ی مرتب‌شده را درنظر می‌گیریم و به ترتیب عناصر دو زیرآرایه را با هم مقایسه می‌کنیم. اگر عنصر کوچکتر در جایگاه بزرگتری قرار گرفته بود، جای آن را با عنصر مورد نظر که در جایگاه کوچکتر است ولی مقدار آن بزرگتر است جابه‌جا می‌کنیم. سپس عناصر را به جلو انتقال می‌دهیم و مرز را هم جابه‌جا می‌کنیم. آنقدر این کار را تکرار می‌کنیم تا سمت راست مرز دیگر عنصری باقی نماند.

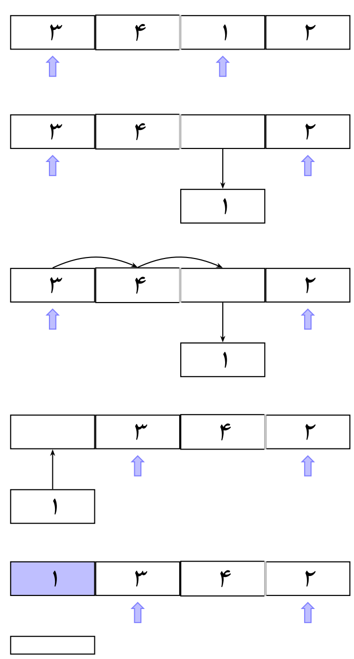
Algorithm 7 INPLACE-MERGE

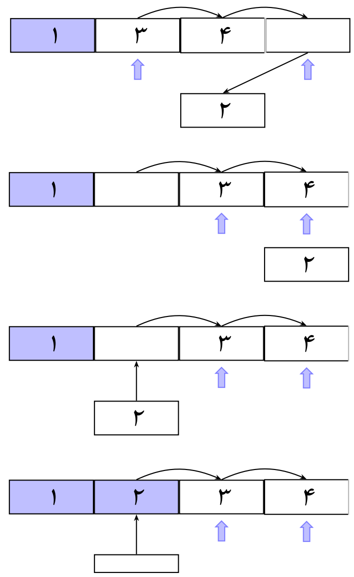
---

function In-Place-Merge($A, low, high$)
  $i \leftarrow low$
  $pivot \leftarrow \frac{low+high}{2}$
  $j \leftarrow pivot + 1$
  while $pivot < high$ and $i \neq j$ **do**
    if $A[i] \leq A[j]$ **then**
      $i \leftarrow i + 1$
    if $A[i] > A[j]$ **then**
      FixArray(i, j)
      $i \leftarrow i + 1$
      $j \leftarrow j + 1$
      $pivot \leftarrow pivot + 1$
function FixArray(i, j)
  $tmp \leftarrow A[j]$
  for $k = j$ **downto** i **do**
    $A[k] \leftarrow A[k-1]$
  $A[i] \leftarrow tmp$

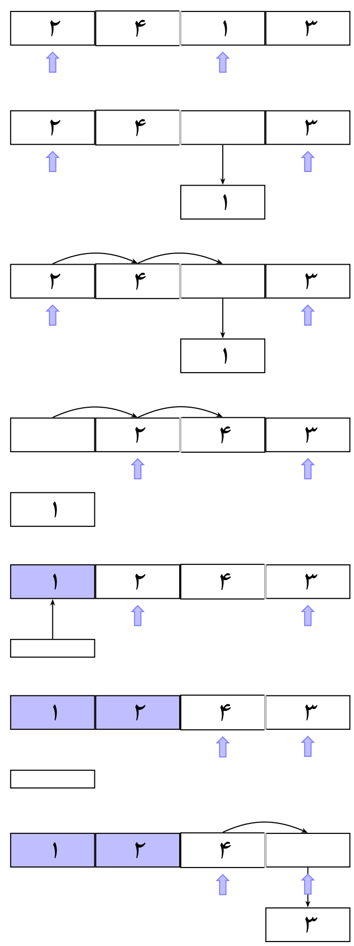
الگوریتم بالا را روی مثال‌های زیر پیاده‌سازی کرده‌ایم:

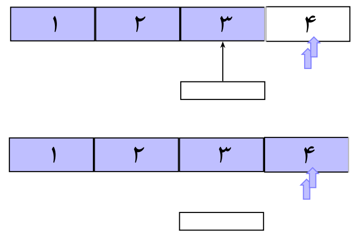
مثال ۱





مثال ۲





توجه داشته باشید با این که تابع InPlaceMerge به صورت درجا عمل می‌کند و تنها به یک واحد حافظه‌ی اضافی جهت جابه‌جایی دو عنصر نیاز دارد، کل الگوریتم مرتب‌سازی ادغامی به کمک تابع InPlaceMerge درجا نیست. زیرا این تابع $\log n$ بار به صورت بازگشتی صدا زده می‌شود. بنابراین این تابع برای اجرا نیاز به $O(\log n)$ واحد حافظه‌ی اضافی نیاز دارد. از آن جایی که این مقدار حافظه وابسته به ورودی است، کل الگوریتم غیر درجا است. البته مقدار حافظه‌ی مورد نیاز آن کمتر از زمانی است که تابع ادغام، درجا نیست.
مرتبه‌ی زمانی کل الگوریتم، $\Theta(n^۲)$ است زیرا داریم:

$$T(n) = ۲T(\frac{n}{۲}) + n^۲ = \Theta(n^۲)$$

اگر کل الگوریتم را به جای بالا به پایین به صورت پایین به بالا پیاده‌سازی کنیم تا دیگر بازگشتی نباشد، به الگوریتم درجای مرتب‌سازی ادغامی می‌رسیم.
نسخه‌ی دیگری از الگوریتم مرتب‌سازی ادغامی وجود دارد که درجاست و پیچیدگی آن $O(n \lg n)$ است ولی به خاطر پیچیده‌بودن از آن خودداری می‌کنیم.

۵.۱ مرتب‌سازی سریع

الگوریتم مرتب‌سازی سریع[۶] خود دارای تابع درجا است ولی چون مانند مرتب‌سازی ادغامی این توابع به صورت بازگشتی صدا زده می‌شوند، مقدار حافظه‌ای که نیاز دارد (علاوه بر حافظه‌ی مورد نیاز برای ذخیره‌ی ورودی) وابسته به تعداد بارهایی است که تابع صدا زده می‌شود؛ یعنی، وابسته به ورودی است که یعنی کل الگوریتم درجا نیست. ولی اگر به جای بالا به پایین آن را به صورت پایین به بالا پیاده سازی کنیم که دیگر بازگشتی نباشد، به یک الگوریتم درجا می‌رسیم.

[۶] Quick Sort