به نام خدا

جستجوی الگوریتم بهینه‌ی مرتب‌سازی در داده‌های مختلف

اردلان سیاوش‌پور[۱]، کسری سیاوش‌پور[۲]، نیما روشن‌زاده[۳] و ارینا مرادی[۴]

[۱] دانشجوی کارشناسی علوم کامپیوتر دانشگاه صنعتی شریف - ۹۹۱۰۹۸۹۶
[۲] دانشجوی کارشناسی علوم کامپیوتر دانشگاه صنعتی شریف - ۹۹۱۰۹۹۰۳
[۳] دانشجوی کارشناسی علوم کامپیوتر دانشگاه صنعتی شریف - ۹۹۱۷۰۹۷۲
[۴] دانشجوی کارشناسی علوم کامپیوتر دانشگاه صنعتی شریف - ۹۹۱۰۷۳۱۳

چکیده: در این مقاله قرار است با توجه به اندازه و نوع داده‌ها، زمان اجرای ۶ الگوریتم معروف مرتب‌سازی برروی آن‌ها را تخمین بزنیم و الگوریتم بهینه را انتخاب و با استفاده از آن، داده‌های ورودی را مرتب کنیم. درنهایت یک جمع‌بندی کلی برروی این الگوریتم‌ها و عملکردشان در مواقع مختلف ارائه می‌دهیم و یک الگوریتم مرتب‌سازی برای انواع داده‌ها ارائه می‌دهیم که در اکثر اوقات به صورت بهینه عمل می‌کند.

کلمات کلیدی: الگوریتم مرتب‌سازی، مرتب‌سازی ادغامی، مرتب‌سازی درجی، مرتبه زمانی

۱ مقدمه

الگوریتم‌های مرتب‌سازی زیادی وجود دارند که هرکدام با توجه به نوع و تعداد داده‌های مسئله می‌توانند کاربردی و اجرایشان منطقی باشد. در این مقاله ۶ الگوریتم مرتب‌سازی: «مرتب‌سازی حبابی (Bubble Sort)، مرتب‌سازی درجی (Insertion Sort)، مرتب‌سازی شمارشی (Counting Sort)، مرتب‌سازی مبنایی (Radix Sort)، مرتب‌سازی ادغامی (Merge Sort) و مرتب‌سازی سریع (Quick Sort)» مورد بررسی قرار می‌گیرند و نکات مثبت و منفی و موارد استفاده از آن‌ها ذکر می‌شود.

```
procedure bubbleSort(A : list of sortable items)
    n := length(A)
    repeat
        swapped := false
        for i := 1 to n-1 inclusive do
            /* if this pair is out of order */
            if A[i-1] > A[i] then
                /* swap them and remember something changed */
                swap(A[i-1], A[i])
                swapped := true
            end if
        end for
    until not swapped
end procedure
```

شکل ۱: شبه‌کد الگوریتم مرتب‌سازی حبابی

۲ بررسی الگوریتم‌ها

۲-۱ مرتب‌سازی حبابی

این الگوریتم از ابتدایی‌ترین الگوریتم‌های مرتب‌سازی است که از مرتبه زمانی $O(n^2)$ و حافظه‌ی اشغالی $O(n)$ است و نسخه‌های بهینه‌یافته‌ی زیادی نیز دارد. الگوریتم اولیه‌ی آن به دلیل اینکه از بسیاری از الگوریتم‌های دیگر (مثل الگوریتم مرتب‌سازی درجی) همواره کندتر است و پیاده‌سازی آن نیز امتیاز خاصی نسبت به الگوریتم‌های دیگر ندارد زیاد استفاده نمی‌شود و امروزه بیشتر یادگیری آن جنبه‌ی آموزشی دارد.

۲-۲ مرتب‌سازی درجی

این الگوریتم یک مرحله از الگوریتم مرتب‌سازی حبابی پیشرفته‌تر است. مرتبه زمانی و حافظه‌ی اشغالی آن با مرتب‌سازی حبابی یکسان است؛ با این حال اردر زمانی آن بسته به ورودی می‌تواند تا $O(n)$ نیز کاهش یابد و در مجموع از ضریب پایین‌تری در اجرا نسبت به مرتب‌سازی حبابی برخوردار است. این الگوریتم به دلیل سرعت نسبتاً خوب برروی داده‌های با تعداد پایین، راحتی و سبکی پیاده‌سازی در داده‌های با تعداد کم بسیار استفاده می‌شود.

```
i ← 1
while i < length(A)
     j ← i
     while j > 0 and A[j-1] > A[j]
          swap A[j] and A[j-1]
          j ← j - 1
     end while
     i ← i + 1
end while
```

شکل ۲: شبه‌کد الگوریتم مرتب‌سازی درجی

```
function CountingSort(input, k)

    count ← array of k + 1 zeros
    output ← array of same length as input

    for i = 0 to length(input) - 1 do
        j = key(input[i])
        count[j] += 1

    for i = 1 to k do
        count[i] += count[i - 1]

    for i = length(input) - 1 downto 0 do
        j = key(input[i])
        count[j] -= 1
        output[count[j]] = input[i]

    return output
```

شکل ۳: شبه‌کد الگوریتم مرتب‌سازی شمارشی

۲-۳ مرتب‌سازی شمارشی

ساز و کار این الگوریتم با سایر الگوریتم‌های نام‌برده فرق دارد و به نوعی فقط برروی داده‌های صحیح و نامنفی استفاده می‌شود و مرتبه زمانی آن $O(n + max(a))$ و حافظه‌ی اشغالی آن $O(max(a))$ است. این الگوریتم در مواقعی که بازه‌ی اعداد داده $(max(a))$ بزرگ نباشد بسیار مناسب است و می‌تواند انتخاب مناسبی باشد.

۲-۴ مرتب‌سازی مبنایی

این الگوریتم داده‌ها را بر حسب یک مبنای مشخص (برای اعداد ۱۰) نگاه می‌کند و آن‌ها را رقم به رقم مرتب می‌کند. مرتبه زمانی آن برابر $O(n \cdot log_b(max(a)))$ و حافظه‌ی اشغالی آن $O(n)$ است. این الگوریتم در زمان‌هایی که تعداد داده‌ها زیاد باشند و یا مقادیر آن‌ها خیلی بزرگ نباشد می‌تواند بسیار خوب عمل کند و همچنین مرتبه زمانی آن می‌تواند با توجه به مقادیر ورودی تا $O(n + log_b(max(a)))$ کاهش یابد.

```
Radix-Sort(A, d)
//It works same as counting sort for d number of passes.
//Each key in A[1..n] is a d-digit integer.
//(Digits are numbered 1 to d from right to left.)
    for j = 1 to d do
        //A[]-- Initial Array to Sort
        int count[10] = {0};
        //Store the count of "keys" in count[]
        //key- it is number at digit place j
        for i = 0 to n do
         count[key of(A[i]) in pass j]++

        for k = 1 to 10 do
         count[k] = count[k] + count[k-1]

        //Build the resulting array by checking
        //new position of A[i] from count[k]
        for i = n-1 downto 0 do
         result[ count[key of(A[i])] ] = A[j]
         count[key of(A[i])]--

        //Now main array A[] contains sorted numbers
        //according to current digit place
        for i=0 to n do
          A[i] = result[i]

    end for(j)
end func
```

شکل ۴: شبه‌کد الگوریتم مرتب‌سازی مبنایی

```
// Sort elements lo through hi (exclusive) of array A.
algorithm mergesort(A, lo, hi) is
    if lo+1 < hi then  // Two or more elements.
        mid := ⌊(lo + hi) / 2⌋
        fork mergesort(A, lo, mid)
        mergesort(A, mid, hi)
        join
        merge(A, lo, mid, hi)
```

شکل ۵: شبه‌کد الگوریتم مرتب‌سازی ادغامی

۲-۵ مرتب‌سازی ادغامی

این مرتب‌سازی از پرکابردترین و سریع‌ترین الگوریتم‌های ممکن برای مرتب‌سازی است. مرتبه زمانی و حافظه‌ی اشغالی آن برابر $O(n \cdot lg(n))$ می‌باشد و روند کار آن بازگشتی است. این الگوریتم در انواع مختلف داده‌ها زمان نسبتاً خوبی برای مرتب‌سازی ارائه می‌دهد و بیشتر در داده‌های با تعداد بالا کاربرد دارد.

۲-۶ مرتب‌سازی سریع

کلیات این الگوریتم مانند الگوریتم مرتب‌سازی ادغامی است. مرتبه زمانی آن به صورت سرشکن از $O(n \cdot lg(n))$ و حافظه‌ی اشغالی آن $O(n)$ است. مرتبه زمانی این الگوریتم می‌تواند بسته به ورودی تا $O(n^2)$ نیز افزایش یابد! به دلیل حافظه‌ی اشغالی کمتر این الگوریتم نسبت به الگوریتم مرتب‌سازی ادغامی، در بعضی موارد استفاده از آن به صرفه‌تر است. این الگوریتم بیشتر در داده‌های با تعداد بالا که ترتیب اولیه‌ی آن‌ها به صورت تصادفی است کاربرد دارد.

Data Type: Double				
نام الگوریتم	$n = 10^2$	$n = 10^3$	$n = 10^4$	$n = 10^5$
مرتب‌سازی درجی	۲۶۶۵.۱	۱۲۷.۱۳	۷۳.۲۶۸	۶.۴۱۶۷۲
مرتب‌سازی مبنایی	۵۱.۰	۱۷.۱	۹۱.۳	۸.۶۸
مرتب‌سازی ادغامی	۱۵.۱	۳۶.۲	۲۴.۶	۸.۹۰
مرتب‌سازی سریع	۵۶.۲	۸۶.۵	۳۶.۹	۳.۴۶
مرتب‌سازی ابداعی	۸۳.۰	۴۳.۲	۹۱.۸	۸.۴۳
مرتب‌سازی جاوا	۶۵.۰	۴۲.۱	۶۵.۶	۰۶.۵۴

Data Type: String				
نام الگوریتم	$n = 10^2$	$n = 10^3$	$n = 10^4$	$n = 10^5$
مرتب‌سازی درجی	۸۶.۰	۸۲.۱۴	۴۶.۴۰۱	۰.۵۵۷۳۹
مرتب‌سازی مبنایی	۸۳.۰	۵۷.۱	۹۴.۶	۳.۳۹
مرتب‌سازی ادغامی	۷۷.۰	۶۵.۲	۴۸.۱۰	۳.۹۸
مرتب‌سازی سریع	۰۵.۳	۳۶.۵	۳۶.۱۲	۸.۵۲
مرتب‌سازی ابداعی	۰۶.۱	۴۶.۲	۵۵.۹	۷.۶۳
مرتب‌سازی جاوا	۴۰.۰	۲۵.۱	۸۲.۸	۴.۷۳

```
// Sorts a (portion of an) array, divides it into partitions, then sorts those
algorithm quicksort(A, lo, hi) is
  // Ensure indices are in correct order
  if lo >= hi || lo < 0 then
    return

  // Partition array and get the pivot index
  p := partition(A, lo, hi)

  // Sort the two partitions
  quicksort(A, lo, p - 1) // Left side of pivot
  quicksort(A, p + 1, hi) // Right side of pivot

// Divides array into two partitions
algorithm partition(A, lo, hi) is
  pivot := A[hi] // Choose the last element as the pivot

  // Temporary pivot index
  i := lo - 1

  for j := lo to hi - 1 do
    // If the current element is less than or equal to the pivot
    if A[j] <= pivot then
      // Move the temporary pivot index forward
      i := i + 1

      // Swap the current element with the element at the temporary pivot index
      swap A[i] with A[j]
  // Move the pivot element to the correct pivot position (between the smaller and larger elements)
  i := i + 1
  swap A[i] with A[hi]
  return i // the pivot index
```

شکل ۶: شبه‌کد الگوریتم مرتب‌سازی سریع

۳ نتیجه‌گیری کلی

در داده‌های کم الگوریتم مرتب‌سازی درجی بهترین انتخاب است، در داده‌های با مقادیر کم الگوریتم مرتب‌سازی مبنایی انتخاب مناسب‌تری است و در داده‌های زیاد الگوریتم مرتب‌سازی ادغامی مناسب است. البته اگر تعداد داده‌ها بالا باشد و به صورت تصادفی پخش شده باشند الگوریتم مرتب‌سازی سریع می‌تواند بهتر عمل کند.

۴ الگوریتم مرتب‌سازی ابداعی

این الگوریتم از روی الگوریتم مرتب‌سازی std::sort() ایده‌گیری شده است و در اصل می‌توان گفت به نوعی فاز اول پیاده‌سازی این الگوریتم به شمار می‌رود. این الگوریتم در داده‌های با تعداد بالا بسیار خوب عمل می‌کند و حتی از الگوریتم مرتب‌سازی جاوا نیز بهتر عمل می‌کند! البته لازم به ذکر است در داده‌های با تعداد پایین می‌تواند از الگوریتم‌هایی مثل مرتب‌سازی درجی کندتر عمل کند.

Data Type: Integer				
نام الگوریتم	$n = 10^2$	$n = 10^3$	$n = 10^4$	$n = 10^5$
مرتب‌سازی درجی	۰۹.۱	۹۲.۱۳	۱۹.۲۷۲	$32564ms$
مرتب‌سازی شمارشی	۱۹.۲۱۹۲	۵۰.۲۱۲۱	۲۱.۲۰۷۵	$2466ms$
مرتب‌سازی مبنایی	۱۰.۰	۸۸.۰	۸۵.۲	$40ms$
مرتب‌سازی ادغامی	۶۸.۰	۳۹.۲	۴۴.۷	$120ms$
مرتب‌سازی سریع	۱۷۷.۳	۸۱.۶	۵۲.۷	$76ms$
مرتب‌سازی ابداعی	۸۵.۰	۸۲.۲	۰۰.۱۰	$65ms$
مرتب‌سازی جاوا	۲۸.۰	۲۴.۱	۴۲.۷	$79ms$