مقایسۀ الگوریتم‌های مرتب‌سازی

محمد ترابی - علی جعفرآبادی - رضا تاج‌گذاری

تیرماه ۱۴۰۲

چکیده

این مقاله به مقایسه الگوریتم‌های مرتب‌سازی پرداخته و عملکرد و ویژگی‌های آن‌ها را مورد بررسی قرار می‌دهد. الگوریتم‌های مرتب‌سازی درجی، انتخابی، ادغامی و سریع به تفصیل بررسی شده و با استفاده از معیارهای پیچیدگی زمانی و حافظه، کارایی آن‌ها مقایسه می‌شود. مزایا و معایب هر الگوریتم به همراه مثال‌ها، قضایا و تجربیات عملی برای نشان دادن عملکرد الگوریتم‌ها بررسی می‌شود. انتخاب یک الگوریتم مرتب‌سازی مناسب برای یک مسئله خاص می‌تواند مقدار زیادی در زمان انجام کار و مصرف منابع صرفه جویی کند. هدف این مقاله، کمک به خوانندگان در انتخاب الگوریتم‌های مرتب‌سازی مناسب برای مسائل مختلف و ارائه یک درک بهتر از الگوریتم‌های مرتب‌سازی است.

• مقدمه

در علوم کامپیوتر، الگوریتم‌های مرتب سازی یکی از موضوعات مهم و پرکاربرد است. این الگوریتم‌ها برای مرتب‌سازی داده‌ها بر اساس قواعد مشخصی طراحی می‌شوند و می‌توانند در حل مسائل گوناگونی مانند جستجو، آمار، پردازش تصویر و ... مورد استفاده قرار گیرند.

در این مقاله، قصد داریم الگوریتم‌های مرتب سازی رایج را مقایسه کنیم. این الگوریتم‌ها شامل مجموعه‌ای از مراحل و محاسبات است که داده‌های ورودی را مرتب می‌کنند. هدف از انجام این مقایسه، شناخت و درک بهتر عملکرد و ویژگی‌های هر الگوریتم است تا بتوانیم در انتخاب و استفاده از الگوریتم‌های مناسب برای مسائل خاص کمک بیشتری به علوم کامپیوتر و صنایع مختلف ارائه دهیم.

در این مقاله، برخی از الگوریتم های مرتب سازی معروف مانند الگوریتم مرتب‌سازی درجی، الگوریتم مرتب‌سازی انتخابی، الگوریتم مرتب‌سازی ادغامی و الگوریتم مرتب‌سازی سریع را مورد بررسی و مقایسه قرار می‌دهیم. به این ترتیب، امید است خوانندگان این مقاله قادر باشند تفاوت‌ها و مزایا و معایب هر الگوریتم را درک کنند و بهترین راه حل را برای مسئله‌ی مورد نظر خود انتخاب کنند.

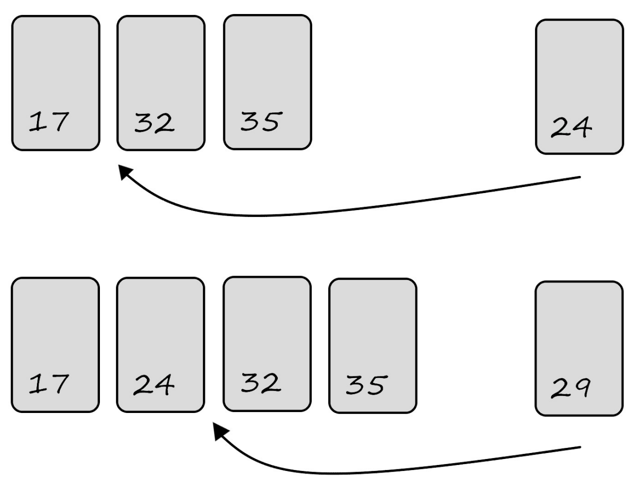
روند مطالب به این صورت است که ابتدا به معرفی هر یک از الگوریتم‌ها پرداخته، سپس با استفاده از معیارهای مختلفی نظیر پیچیدگی زمانی و حافظه، کارایی آن‌ها را بررسی خواهیم کرد. همچنین مثال‌ها، قضایا و تجربیاتی عملی برای نشان دادن عملکرد الگوریتم‌ها ارائه خواهیم داد.

امیدواریم که این مقاله به خوانندگان کمک کند تا در انتخاب و استفاده از الگوریتم‌های مرتب سازی برای مسائل خود بهتر عمل کنند و درک بهتری از این الگوریتم‌های مرتب‌سازی به دست آورند.

۱ مرتب‌سازی درجی[1]

اگر یک دسته کارت به شما داده شود که اعداد ۱ تا ۵۰ روی آن نوشته شده است، چگونه آن را مرتب[2] می‌کنید؟ احتمالا اول تعداد کمی کارت برمی‌دارید و آن‌ها را مرتب می‌کنید؛ سپس بقیه کارت‌ها را یکی پس از دیگری نگاه می‌کنید و در جای مناسب میان کارت‌های مرتب شده قرار می‌دهید. شکل ۱ نمایی کلی از این روش مرتب‌سازی نشان می‌دهد.

وقتی کارت‌ها را با این روند مرتب می‌کنیم، همواره تعدادی از کارت‌ها مرتب شده است و کارت‌هایی که هنوز مرتب نشده، یکی پس از دیگری در دستهٔ کارت‌های مرتب شده درج[3] می‌شوند. اگر با این روش کارت‌ها را مرتب کنیم، درواقع از مرتب‌سازی درجی استفاده کرده‌ایم.



شکل ۱: مراحلی از مرتب کردن کارت‌ها به کمک مرتب‌سازی درجی

[1] Insertion sort
[2] sort
[3] insert

الگوریتم مرتب‌سازی درجی

آرایه‌ای به طول یک همواره مرتب است؛ بنابراین از عنصر دوم شروع می‌کنیم و جلو می‌رویم. فرض کنیم که به عنصر i رسیده‌ایم، عناصر قبل از i مرتب‌اند. لذا از آخرین عنصر قبل از i شروع می‌کنیم و به عقب می‌رویم. هر یک از عناصر بزرگ‌تر از i را یک واحد به سمت راست انتقال[۴] می‌دهیم. وقتی به عنصری کوچک تر از i یا به ابتدای آرایه رسیدیم متوقف می‌شویم و i را همانجا درج می‌کنیم. هنگامی که عنصر i را درج می‌کنیم، i عنصر اول آرایه مرتب می‌شوند. بنابراین این کار را ادامه می‌دهیم تا تمام عناصر آرایه مرتب شوند. درستی مرتب‌سازی درجی را می‌توان به کمک ثابت‌های حلقه[۵] اثبات کرد. [۱] شبه‌کد مرتب‌سازی درجی را می‌توانید در الگوریتم ۱ که در ادامه آمده است مشاهده کنید [۶] .

الگوریتم ۱ مرتب‌سازی درجی

```
procedure InsertionSort(arr, n)
    for i ← 1 to n − 1 do
        key ← arr[i]
        j ← i − 1
        while j >= 0 and arr[j] > key do
            arr[j + 1] ← arr[j]
            j ← j − 1
        arr[j + 1] ← key
```

مثال ۱ *اگر الگوریتم بالا را روی آرایهٔ $[۷, ۶, ۶, ۴, ۹]$ اجرا کنیم، آرایه بدین صورت مرتب می‌شود:*

$$[۷, ۶, ۶, ۴, ۹] \rightarrow [۷, ۷, ۶, ۴, ۹] \rightarrow [۶, ۷, ۶, ۴, ۹] \rightarrow [۶, ۷, ۷, ۴, ۹] \rightarrow$$
$$[۶, ۶, ۷, ۴, ۹] \rightarrow [۶, ۶, ۷, ۷, ۹] \rightarrow [۶, ۶, ۶, ۷, ۹] \rightarrow [۶, ۶, ۶, ۷, ۹] \rightarrow$$
$$[۴, ۶, ۶, ۷, ۹] \rightarrow [۴, ۶, ۶, ۷, ۹]$$

[4] shift

[5] loop invariants

[۶] توجه داشته باشید که الگوریتم بر پایه صفر است؛ یعنی عنصر اول آرایه $arr[۰]$ می‌باشد.

توجه داشته باشید که پیچیدگی زمانی مرتب‌سازی درجی در بهترین حالت خطی است. همچنین بهترین حالت وقتی اتفاق می‌افتد که آرایه مرتب باشد. می‌توانیم نتیجه بگیریم که در مواقعی که آرایه مرتب و یا تقریبا مرتب است مرتب‌سازی درجی بسیار سریع عمل می‌کند. بنابراین اگر از قبل مطلع هستیم که معمولا داده‌های ما تقریبا مرتب است، استفاده از مرتب‌سازی درجی می‌تواند گزینه مناسبی باشد.

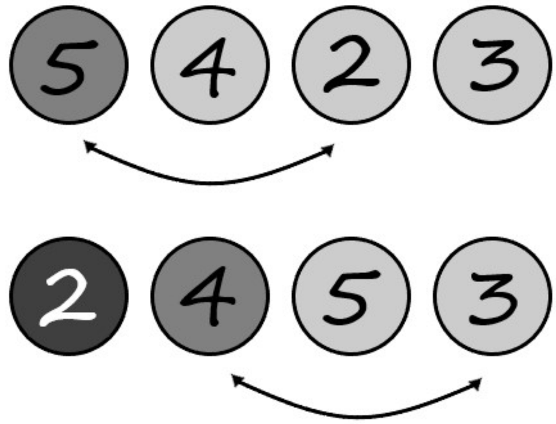
به عنوان مثال فرض کنید که یک لیست هزارتایی از اسامی دانشجویان یک موسسه در اختیار دارید که به ترتیب حروف الفبا مرتب شده اند. در سال جدید پنجاه دانشجو در موسسه نام نویسی می‌کنند و اسامی آن‌ها به آخر آرایه اضافه می‌شود. اگرچه روش‌های زیادی برای مرتب کردن لیست جدید دانشجویان وجود دارد، مرتب‌سازی درجی گزینه مناسبی محسوب می‌شود و از دیگر روش‌های مرتب‌سازی که در این مقاله توضیح داده شده، بهتر عمل می‌کند.

خوب است بدانید می‌توان پیچیدگی زمانی مرتب سازی درجی در بدترین حالت را با تغییراتی در الگوریتم آن بهبود بخشید. به عنوان مثال مرتب‌سازی شِل[۷] یک روش مرتب‌سازی دیگر است که از مرتب‌سازی درجی استفاده می‌کند و در حالت کلی دارای پیچیدگی زمانی بهتری است. [۲] [۳] جزئیات الگوریتم و پیچیدگی زمانی مرتب‌سازی شل را در این مقاله بررسی نمی‌کنیم.

۲ مرتب‌سازی انتخابی[۸]

مرتب‌سازی انتخابی الگوریتم ساده‌ای دارد و احتمالا یکی از اولین الگوریتم‌های مرتب‌سازی باشد که به ذهنمان می‌رسد. مرتب‌سازی انتخابی اینگونه عمل می‌کند که کوچک‌ترین عنصر آرایه را انتخاب می‌کند، آن را در سمت چپ آرایه قرار می‌دهد و سپس به سراغ عناصر باقی‌مانده می‌رود. [۱] برای درک بهتر می‌توانید به شکل ۲ که در ادامه آمده است نگاه کنید.

[7] Shell sort
[8] Selection sort



شکل ۲: مراحلی از مرتب کردن اعداد به کمک مرتب‌سازی انتخابی

الگوریتم مرتب‌سازی انتخابی

از اولین عنصر آرایه شروع می‌کنیم. اگر به عنصر i برسیم، i را به عنوان اندیسِ[9] کوچک‌ترین عنصر ذخیره می‌کنیم؛ سپس از عنصر بعد از i شروع می‌کنیم و جلو می‌رویم. اگر عنصری از کوچک‌ترین عنصر کوچک‌تر بود اندیس آن را به عنوان اندیس کوچک‌ترین عنصر نگه می‌داریم. وقتی به آخر آرایه رسیدیم، کوچک‌ترین عنصر ذخیره شده را با i جابه‌جا[10] می‌کنیم و به سراغ عنصر بعدی می‌رویم. مراحل گفته شده را برای i جدید تکرار می‌کنیم تا به انتهای آرایه برسیم. شبه‌کد مرتب‌سازی انتخابی در الگوریتم ۲ که در ادامه آمده است قابل مشاهده است. همچنین شبه‌کد جابه‌جایی را می‌توانید در الگوریتم ۳ مشاهده کنید.

[9] index
[10] swap

الگوریتم ۲ مرتب‌سازی انتخابی

```
procedure SelectionSort(arr, n)
    for i ← 0 to n − 1 do
        minIndex ← i
        for j ← i + 1 to n do
            if arr[j] < arr[minIndex] then
                minIndex ← j
        Swap(arr, i, minIndex)
```

الگوریتم ۳ جابه‌جایی

```
procedure Swap(arr, i, j)
    temp ← arr[i]
    arr[i] ← arr[j]
    arr[j] ← temp
```

مثال ۲ *اگر الگوریتم ۲ را روی آرایهٔ $[۵, ۴, ۲, ۳, ۷]$ اجرا کنیم، آرایه بدین صورت مرتب می‌شود:*

$$[۵, ۴, ۲, ۳, ۷] \rightarrow [۲, ۴, ۵, ۳, ۷] \rightarrow [۲, ۳, ۵, ۴, ۷] \rightarrow [۲, ۳, ۴, ۵, ۷] \rightarrow [۲, ۳, ۴, ۵, ۷]$$

مثال ۳ *اگر الگوریتم ۲ را روی آرایهٔ $[۷, ۶, ۶, ۴, ۹]$ اجرا کنیم، آرایه بدین صورت مرتب می‌شود:*

$$[۷, ۶, ۶, ۴, ۹] \rightarrow [۴, ۶, ۶, ۷, ۹] \rightarrow [۴, ۶, ۶, ۷, ۹] \rightarrow [۴, ۶, ۶, ۷, ۹] \rightarrow [۴, ۶, ۶, ۷, ۹]$$

نکته ۱ *توجه کنید در مثال ۳ با اینکه آرایه بعد از یک جابه‌جایی مرتب می‌شود، الگوریتم به کار خود ادامه می‌دهد و به‌ازای تمام عناصر i تا $n - ۱$ اجرا می‌شود. با مقایسه مثال ۲ و مثال ۳ که در بالا آمده اند، می‌توان وابسته نبودن مرتب‌سازی انتخابی به عناصر ورودی را بهتر درک کرد.*

۳ مرتب‌سازی ادغامی[11]

مرتب‌سازی ادغامی یکی دیگر از روش‌های مرتب سازی است که از روش تقسیم و حل[12] استفاده می‌کند. این روش در سال ۱۹۴۵ میلادی ابداع شد. [۴] استفاده از این روش نسبت به روش‌های قبلی سرعت مرتب شدن اعداد را به طور قابل توجهی بالا می‌برد.

این روش اینگونه عمل می‌کند که آرایه را به دو قسمت تقسیم می‌کند و هر قسمت را به صورت بازگشتی مرتب می‌کند. وقتی که دو قسمت آرایه مرتب شدند، در واقع دو آرایه مرتب شده داریم و می‌خواهیم آن دو را باهم ادغام[13] کنیم. ادغام کردن دو آرایه مرتب شده و تبدیل آن به یک آرایه مرتب شده، در زمان خطی[14] قابل انجام است. خطی بودن ادغام یکی از دلایل سریع شدن مرتب‌سازی درجی است.

الگوریتم مرتب‌سازی ادغامی

آرایه اولیه را به دو قسمت، که اندازه هر قسمت حدود نصف آرایه اولیه است، تقسیم می‌کنیم. این عمل را به طور بازگشتی ادامه می‌دهیم تا به آرایه‌هایی به یک عضو برسیم. آرایه با یک عضو همواره مرتب است. سپس آرایه‌های کوچک‌تر مرتب شده را با یکدیگر ادغام می‌کنیم تا آرایه‌های بزرگتر مرتب شده داشته باشیم. این عمل را روی کل آرایه انجام می‌دهیم و کل آرایه مرتب می‌شود. شبه‌کد مرتب‌سازی ادغامی و شبه‌کد ادغام[15] ، که در مرتب‌سازی ادغامی استفاده می‌شود، به ترتیب در الگوریتم ۴ و الگوریتم ۶ در ادامه آمده است.

---

[11] Merge sort
[12] divide and conquer
[13] merge
[14] linear
[15] merge

الگوریتم ۴ مرتب‌سازی ادغامی

procedure MERGESORT($arr, left, right$)
    if $left < right$ **then**
        $middle \leftarrow \frac{left+right}{2}$
        MERGESORT($arr, left, middle$)
        MERGESORT($arr, middle + 1, right$)
        MERGE($arr, left, middle, right$)

الگوریتم ۵ ادغام

```
procedure Merge(arr, left, middle, right)
    n1 ← middle − left + 1
    n2 ← right − middle
    leftArr ← new Array[n1]
    rightArr ← new Array[n2]
    for i ← 0 to n1 do
        leftArr[i] ← arr[left + i]
    for j ← 0 to n2 do
        rightArr[j] ← arr[middle + 1 + j]
    i ← 0
    j ← 0
    k ← left
    while i < n1 and j < n2 do
        if leftArr[i] ≤ rightArr[j] then
            arr[k] ← leftArr[i]
            i ← i + 1
        else
            arr[k] ← rightArr[j]
            j ← j + 1
        k ← k + 1
    while i < n1 do
        arr[k] ← leftArr[i]
        i ← i + 1
        k ← k + 1
    while j < n2 do
        arr[k] ← rightArr[j]
        j ← j + 1
        k ← k + 1
```

مثال ۴ *اگر الگوریتم ۴ را روی آرایهٔ* $[۷, ۶, ۳, ۱]$ *اجرا کنیم، آرایه بدین صورت مرتب می‌شود:*

$$[۷, ۶, ۳, ۱] \rightarrow [۷, ۶], [۳, ۱] \rightarrow [۷], [۶], [۳, ۱] \rightarrow [۶, ۷], [۳, ۱] \rightarrow [۶, ۷], [۳], [۱] \rightarrow [۶, ۷], [۱, ۳] \rightarrow [۱, ۳, ۶, ۷]$$

***نکته ۲** مرتب‌سازی ادغامی نسبت به مرتب‌سازی‌های دیگری که در این مقاله بحث شده به حافظه بیشتری نیاز دارد. البته قابل توجه است که اقداماتی برای تغییر این الگوریتم با این هدف که پیچیدگی حافظه آن کم شود انجام شده. چندین روش برای مرتب‌سازی ادغامی با حافظه ثابت[16] یا به عبارتی مرتب سازی ادغامیِ درجا[17] پیشنهاد شده است. [۵]*

۴ مرتب‌سازی سریع[18]

مرتب‌سازی سریع یکی از روش‌های پراستفاده برای مرتب‌سازی است. این روش نه‌تنها سرعت بالایی دارد، بلکه از نظر حافظه نیز خوب عمل می‌کند و حافظه نسبتا کمی استفاده می‌کند. یکی دیگر از مزایای مرتب سازی سریع این است که الگوریتم نسبتا ساده‌ای دارد و پیاده سازی آن دشوار نیست. این الگوریتم توسط تونی هور[19] در سال ۱۹۵۹ میلادی ابداع شد.[۶] این روش به طور کلی، در مرتب کردنِ داده‌های تصادفی، از مرتب‌سازی ادغامی سریع‌تر عمل می‌کند؛ مخصوصا در داده‌های بزرگ‌تر.[۷]

مرتب‌سازی سریع نیز همانند مرتب‌سازی ادغامی از روش تقسیم و حل[20] استفاده می‌کند. ابتدا یک عنصر محوری[21] انتخاب می‌کند؛ سپس عناصر کوچک‌تر را سمت چپ و عناصر بزرگ‌تر را در سمت راست آن قرار می‌دهد. به همین دلیل این روش، مرتب‌سازی بخش‌کردن‌ـ تبادل[22] نیز نامیده می‌شود.[۸]

الگوریتم مرتب‌سازی سریع

مرتب سازی سریع به این شکل عمل می‌کند که ابتدا یک عنصر محوری را انتخاب می‌کنیم .سپس لیست را به دو بخش تقسیم می‌کنیم، یک بخش برای عناصری که

[16] constant
[17] in-place merge sort
[18] Quicksort
[19] Tony Hoare
[20] divide and conquer
[21] pivot
[22] partition-exchange sort

کوچکتر یا مساوی عنصر محوری هستند و دیگری برای عناصری که بزرگتر از عنصر محوری هستند. عنصر محوری می‌تواند هر عنصری باشد مثلا برای راحتی می‌توانیم عنصر اول آرایه را در نظر بگیریم. ولی برای اینکه پیچیدگی زمانی الگوریتم بهینه شود و الگوریتم سریع‌تر عمل کند، این عنصر را با استفاده از الگوریتم دیگری به نام بخش کردن[۲۳] انتخاب می‌کنیم.

پس از انتخاب عنصر محوری و تقسیم آرایه به دو بخش، این دو بخش را به طور مستقل مرتب می‌کنیم، با این معنی که برای هر بخش، عنصر محوری جدیدی را انتخاب می‌کنیم و عملیات تقسیم و حل را بر روی آن اجرا می‌کنیم. این فرایند تا زمانی ادامه می‌یابد که دیگر نتوان بخش‌ها را به زیربخش‌هایی کوچکتر تقسیم کرد. در انتها، وقتی که همه بخش‌ها به صورت مرتب شده باشند، آنها را با هم ترکیب می‌کنیم و آرایه به صورت کامل مرتب می‌شود. شبه‌کد مرتب سازی سریع و بخش کردن را به ترتیب در ادامه در الگوریتم ۶ و الگوریتم ۷ مشاهده می‌کنید.

الگوریتم ۶ مرتب‌سازی سریع

```
procedure QuickSort(arr, low, high)
    if low < high then
        pivot ← Partition(arr, low, high)
        QuickSort(arr, low, pivot − 1)
        QuickSort(arr, pivot + 1, high)
```

الگوریتم ۷ بخش کردن

```
procedure Partition(arr, low, high)
    pivot ← arr[high]
    i ← low − 1
    for j ← low to high − 1 do
        if arr[j] ≤ pivot then
            Swap arr[i + 1] and arr[j]
            i ← i + 1
    Swap arr[i + 1] and arr[high]
    return i + 1
```

[23] partition

مثال ۵ *فرض کنید می‌خواهیم آرایهٔ [۱, ۴, ۲, ۳] را به کمک مرتب‌سازی سریع مرتب کنیم. ابتدا یک عنصر محوری انتخاب می‌کنیم. مثلا اولین عنصر که ۳ است. عناصر کوچک‌تر یعنی ۲ و ۱ را به سمت چپ و عناصر بزرگ‌تر یعنی ۴ را به سمت راست می‌بریم. آرایه به شکل [۴, ۳, ۱, ۲] درمی‌آید. در سمت چپِ ۳، [۲, ۱] و در سمت راست آن [۴] را داریم. [۴] یک عنصر دارد و مرتب شده است. برای مرتب کردنِ [۲, ۱] دوباره مراحل بالا را تکرار می‌کنیم. ۲ را به عنوان عنصر محوری انتخاب می‌کنیم و عناصر کوچک تر از آن یعنی ۱ را به سمت چپ آن می‌بریم. آرایه مرتب می‌شود و به صورت [۱ ، ۲] درمی‌آید. وقتی به آرایه کلی نگاه می‌اندازیم مشاهده می‌کنیم که به صورت [۴, ۳, ۲, ۱] است و مرتب شده است.*

نکته ۳ *برای بهینه‌سازی پیچیدگی زمانی و پیچیدگی حافظهٔ مرتب‌سازی سریع نیز روش‌هایی وجود دارد. مثلا می‌توان ابتدا بخش کوچک‌تر را مرتب کرد و سپس با فراخوانی دم[24] بخش بزرگتر را مرتب کرد تا پیچیدگی حافظه بهتر شود. یا می‌توان وقتی که بخش‌ها کوچک شد (مثلا به کمتر از ۱۰ عنصر رسید) از یک روش مرتب‌سازی غیر بازگشتی مثل مرتب‌سازی درجی استفاده کنیم تا الگوریتم بهینه‌تری داشته باشیم.*[۹][۱۰]

[24] tail call

مراجع

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 4th ed. , 2022.

[2] R. B. Frank, R. M.; Lazarus, "A high-speed sorting procedure," *Communications of the ACM*, p.20–22, 1960.

[3] R. Sedgewick, "A new upper bound for shellsort," *Journal of Algorithms*, p.159–173, 1986.

[4] D. Knuth. *The Art of Computer Programming*. Addison-Wesley, 2nd ed. .

[5] M. A. Huang, Bing-Chao; Langston, "Practical in-place merging," *Communications of the ACM*, p.348–352, 1988.

[6] C. A. R. Hoare, "Algorithm 64: Quicksort," *Comm. ACM*, p.321, 1961.

[7] S. S. Skiena, "The algorithm design manual," *Springer*, p.129, 2008.

[8] C. Foster, "Algorithms, abstraction and implementation," p.98, 1992.

[9] R. Sedgewick, "Implementing quicksort programs," *Comm. ACM.*, p.847–857, 1978.

[10] R. E. LaMarca, Anthony; Ladner, "The influence of caches on the performance of sorting," *LaMarca, Anthony; Ladner, Richard E.*, p.66–104, 1999.