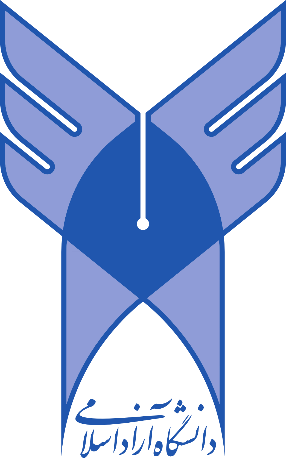
**C:\Program Files (x86)\Hamoon Soft\Chalipa\Chalipa Clipart\Brdr01.wmf**

دانشگاه آزاد اسلامی تهران مرک

مجتمع آیت الله هاشمی رفسنجانی

عنوان :

**تحلیل پیچیدگی الگوریتم ها**

رشته تحصیلی :

**مهندسی کامپیوتر نرم افزار**

استاد راهنما:

**سرکارخانم دکتر نیره زاغری**

تهیه کنندگان:

**حسن پرسته**

**مجتبی سیانکی**

**زمستان 99**

**C:\Program Files (x86)\Hamoon Soft\Chalipa\Chalipa Clipart\Brdr01.wmf**

**الگوریتم های مرتب سازی**

منظور از مرتب سازی داده، چیدمان داده ‌ها در قالبی خاص است. الگوریتم مرتب سازی روشی برای چیدمان داده ‌ها با ترتیبی خاص تعیین می ‌کند. اغلب ترتیب های رایج به صورت ترتیب عددی یا الفبای هستند. اهمیت مرتب‌ سازی در این نکته است که جستجوی داده ‌ها در صورت مرتب بودن می ‌تواند تا سطح بالایی بهینه سازی شود. مرتب‌ سازی همچنین می ‌تواند برای نمایش داده ‌ها در قالب ‌های خواناتر کمک کند.

**اصطلاح‌ های مهم**

برخی اصلاح‌ ها به طور کلی در مباحث مرتبط با تکنیک‌ های مرتب ‌سازی بیشتر به چشم می ‌آیند که در ادامه برخی از آن‌ ها را ارائه کرده‌ ایم:

**ترتیب صعودی:** یک توالی از مقادیر مختلف، زمانی دارای ترتیب صعودی نامیده می ‌شود که همه عناصر پشت سر هم بزرگ ‌تر از عنصر قبلی ‌شان باشند. برای نمونه فهرست 1، 3، 4، 8، 9 دارای ترتیب صعودی است چون هر عنصر از عنصر قبلی خود بزرگ ‌تر است.

**ترتیب نزولی:** یک توالی از مقادیر، زمانی دارای ترتیب نزولی خوانده می ‌شود که همه عناصر پشت سر هم از عنصر قبلی خود کوچک ‌تر باشند. برای نمونه، 9، 8، 6، 4، 3، 1 دارای ترتیب نزولی است و هر عنصر از عنصر قبلی خود کوچک ‌تر است.

**ترتیب غیر افزایشی:** یک توالی از مقادیر دارای ترتیب غیر افزایشی نامیده می ‌شود، وقتی همه عناصر پشت سر هم کمتر یا مساوی عنصر قبلی‌ شان باشند. این ترتیب زمانی رخ می‌ دهد که در توالی مقادیر مورد نظر، عناصری تکراری وجود داشته باشند. برای نمونه، 9، 8، 6، 3، 3، 1 دارای ترتیب غیر افزایشی است، چون هر عنصر کوچک ‌تر یا مساوی (در مورد 3) از عنصر قبلی است، اما هیچ عنصری از عنصر قبلی خود بزرگ‌ تر نیست.

**ترتیب غیر کاهشی:** یک توالی از مقادیر، دارای ترتیب غیر کاهشی خوانده می ‌شود، اگر عناصر پشت سر هم همواره بزرگ ‌تر یا مساوی عنصر قبلی ‌شان باشند. این ترتیب زمانی رخ می ‌دهد که در یک توالی عناصر تکراری وجود داشته باشد. برای نمونه لیست 1، 3، 3، 6، 8، 9 یک فهرست با ترتیب غیر کاهشی است، چون هر عنصر کوچک ‌تر یا مساوی (در مورد 3) از عنصر قبلی است؛ اما بزرگ ‌تر نیست.

**در ادامه الگوریتم مرتب سازی حبابی را مشاهده می کنید!**

**الگوریتم مرتب ‌سازی حبابی (Bubble Sort)**

مرتب ‌سازی حبابی یک الگوریتم مرتب ‌سازی ساده است. این الگوریتم مرتب‌ سازی یک الگوریتم مبتنی بر مقایسه است که در آن هر جفت از عناصر مجاور با هم مقایسه می ‌شوند و در صورتی که در ترتیب مطلوب نباشند با هم تعویض می ‌شوند. این الگوریتم برای مجموعه داده‌ های بزرگ مطلوب نیست، زیرا پیچیدگی حالت میانگین و بدترین حالت آن برابر با است که در آن تعداد آیتم ‌هایی است که باید مرتب شوند.

* **ویژگی‌های مرتب‌سازی حبابی**

1. مرتب‌سازی حبابی یک روش مرتب‌سازی درجا است. یعنی نیاز به فضای کمکی نداشته و با جابجا کردن عناصر در داخل خود لیست، آنها را مرتب می‌کند.
2. مرتب‌سازی حبابی یک روش مرتب‌سازی پایدار است. یعنی در حین مرتب‌سازی ترتیب عناصری که مقدار یکسانی دارند تغییر نمی‌کند.
3. پیچیدگی زمانی آن به صورت است.

**طرز کار مرتب سازی حبابی:**

به عنوان مثال یک آرایه نامرتب را در نظر می‌ گیریم. الگوریتم مرتب ‌سازی حبابی باید مرتبه روی این آرایه کار کند تا آن را مرتب و منظم کند.

۱. ابتدا، دو عنصر اول آرایه با یکدیگر مقایسه می‌شوند و با توجه به آنکه ۵ از ۱ بزرگتر است (۱<۵)، این دو عنصر با یکدیگر جا به جا می‌شوند.

( 5 1 4 2 8 ) –> ( 1 5 4 2 8 )

۲. در اینجا، عناصر دوم و سوم آرایه مقایسه می‌شوند و با توجه به اینکه ۵ از ۴ بزرگ‌تر است (۴<۵)، این دو عنصر با یکدیگر جا به جا می‌شوند.

( 1 5 4 2 8 ) –> ( 1 4 5 2 8 )

۳. اکنون، عنصر سوم و چهارم آرایه مقایسه می‌شوند و با توجه به اینکه ۲ از ۵ کوچک‌تر است (۲<۵)، این دو عنصر با یکدیگر جا به جا می‌شوند.

( 1 4 5 2 8 ) –> ( 1 4 2 5 8 )

۴. در اینجا، عنصر چهارم و پنجم آرایه مقایسه می‌شود و چون ۵ از ۸ کوچک‌تر است (۵<۸) دو عنصر در جای خود بدون هر گونه جا به جایی باقی می‌مانند؛ چون در واقع، ترتیب (صعودی) در آن‌ها رعایت شده است.

( 1 4 2 5 8 ) –> ( 1 4 2 5 8 )

اکنون یک دور کامل در آرایه زده شد. دومین دور نیز به شیوه بیان شده در بالا انجام می‌شود.

۱. جا به جایی اتفاق نمی‌افتد.

( 1 4 2 5 8 ) –> ( 1 4 2 5 8 )

۲. با توجه به بزرگ‌تر بودن ۴ از ۲ (۲<۴)، این دو عنصر با یکدیگر جا به جا می‌شوند.

( 1 4 2 5 8 ) –> ( 1 2 4 5 8 )

۳. جا به جایی اتفاق نمی‌افتد.

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

۴. جا به جایی اتفاق نمی‌افتد.

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

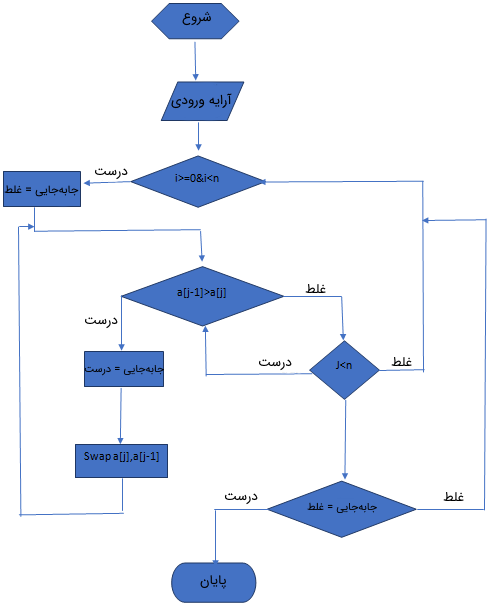
در حال حاضر، آرایه مرتب شده است، اما الگوریتم نمی‌داند که آیا کار به پایان رسیده یا خیر؛ بنابراین، به یک دور کامل دیگر بدون انجام هرگونه جا به جایی نیاز دارد تا بفهمد که مرتب‌سازی با موفقیت به پایان رسیده است.

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

فلوچارت الگوریتم مرتب سازی حبابی:



**کد الگوریتم مرتب سازی حبابی به C++:**

****

**محاسبه ی پیچیدگی زمانی در بهترین ، متوسط و بدترین حالت:**

* **بهترین حالت**
* **حالت متوسط**
* **بدترین حالت**

حال چگونه این ها محاسبه شدند؟

**توضیح اجرای الگوریتم:**

فرض کنیم لیستی با n عنصر با روش مرتب‌سازی حبابی مرتب می‌شوند. عمل اصلی روش‌های مرتب‌سازی مبتنی بر مقایسه، مقایسه‌ها هستند. در مرحله‌ی اول، n - 1 مقایسه صورت می‌گیرد، تا بزرگترین عنصر به انتهای لیست منتقل شود. در مرحله‌ی دوم، n - 2 مقایسه و به همین ترتیب، در مرحله‌ی iام (i < n) تعداد n - i مقایسه صورت می‌گیرد. این تعداد را می‌توان از کد برنامه‌نویسی ذکر شده‌ی فوق هم استخراج کرد. پس تعداد کل مقایسه‌ها برای مرتب کردن n عنصر برابر است با:

چنین عبارتی از مرتبه ی است.

بهینه سازی الگوریتم مرتب سازی حبابی؟

برای بهینه‌تر شدن الگوریتم، کدها را به صورت زیر تغییر می‌دهیم:

void bubble\_sort\_2(int *arr*[], int *n*)

{

    int i, j, t, c, hasChange;

    for (i = *n* - 2; i >= 0; i--)

    {

        hasChange = 0;

        for (j = 0; j <= i; j++)

            if (*arr*[j] > *arr*[j + 1])

            {

                t = *arr*[j];

*arr*[j] = *arr*[j + 1];

*arr*[j + 1] = t;

                hasChange = 1;

            }

        if (hasChange == 0)

            break;

    }

}

مقدار متغیر hasChange قبل از شروع هر مرحله صفر شده و در صورت جابجا شدن محتوای دو خانه از آرایه مقدار آن به یک تغییر پیدا می‌کند. بنابراین، اگر مقدار آن در پایان مرحله صفر باشد، یعنی هیچگونه جابجایی در لیست صورت نگرفته و می‌توان نتیجه گرفت لیست مرتب است (چرا؟).

چنین کدی مرتبه‌ی زمانی اجرای الگوریتم را در بهترین حالت به تقلیل می‌دهد. چرا که اگر لیست از همان ابتدا مرتب باشد، با تمام شدن اولین مرحله (با n-1 مقایسه) و بررسی متغیر C ، مرتب بودن لیست مشخص شده و ادامه روند مرتب‌سازی متوقف می‌شود.

**ویژگی های مرتب سازی حبابی**

1. **همانگونه که بحث شد، پیچیدگی زمانی اجرای این الگوریتم در بدترین حالت و حالت متوسط از مرتبه‌ی است. پیچیدگی زمانی بهترین حالت، با تابع مرتب سازی اول (بهینه نشده) از مرتبه ی و با تابع مرتب سازی دوم (نسخه ی بهینه شده) از مرتبه ی است.**
2. **مرتب‌سازی حبابی یک روش مرتب‌سازی درجا است. یعنی نیاز به فضای کمکی نداشته و با جابجا کردن عناصر در داخل خود لیست، آنها را مرتب می‌کند.**
3. **مرتب‌سازی حبابی - با پیاده‌سازی به یکی از روش‌های فوق - یک روش مرتب‌سازی پایدار است. یعنی در حین مرتب‌سازی ترتیب عناصری که مقدار یکسانی دارند تغییر نمی‌کند. اگر در قطعه کدهای فوق، در مقایسه‌ی عناصر آرایه به جای < از =< استفاده می‌کردیم، مرتب‌سازی ناپایدار می‌شد. چرا که عناصری با مقادیر یکسان را نیز جابجا کرده و ترتیب آنها را به هم می‌زد.**

**مرتب سازی درجی (Insertion Sort)**

روش مرتب‌سازی درجی (Insertion Sort) یکی از روش‌های مقدماتی مرتب‌سازی مبتنی بر مقایسه‌ی عناصر است که در مقایسه با روش‌های دیگر بیشتر مورد توجه قرار دارد.

قفسه‌ی کتابی را در نظر بگیرید که قصد دارید کتاب‌ها را بر اساس عنوان و به ترتیب حروف الفبا مرتب کنید. از یک سمت قفسه شروع به مرتب کردن می‌کنید. ابتدا کتاب دوم را با کتاب اول مقایسه کرده و در صورت لزوم جابجا می‌کنید. سپس کتاب سوم را از محل خود برداشته و در مقایسه با دو کتاب قبلی در محل مناسب قرار می‌دهید. به همین ترتیب کتاب‌های بعدی را نیز نسبت به کتاب‌های مرتب‌شده‌ی قبلی در محل مناسب درج می‌کنید تا به آخر قفسه برسید.

این الگوریتم برای تعداد داده‌های زیاد، کارآمد نیست و در این موارد، الگوریتم‌های بهتری مثل مرتب‌ساز سریع، مرتب‌ساز ادغامی و مرتب‌ساز پشته وجود دارد.

عملکرد این الگوریتم به گونه‌ای است که در پایان هر مرحله قسمتی از داده‌ها به صورت کامل مرتب هستند. در مرحله‌ی بعدی نیز داده‌ای از میان داده‌های غیرمرتب به این قسمت مرتب وارد شده و در محل مناسب درج می‌شود. اگر بخواهیم با روش مرتب‌سازی درجی لیست اعداد زیر را به صورت صعودی (کوچک به بزرگ) مرتب ‌کنیم، در پایان هر مرحله ترتیب عناصر به صورت زیر خواهد بود:

0)    5 1 2 7 3 6

1)    **1 5** 2 7 3 6

2)    **1 2 5** 7 3 6

3)    **1 2 5 7** 3 6

4)    **1 2 3 5 7** 6

5)    **1 2 3 5 6 7**

**مرتب سازی درجی به زبان ساده:**

ایده پشت مرتب سازی درجی این است که در حین مرتب سازی آرایه به دو بخش عناصر مرتب شده و عناصر مرتب نشده تقسیم شود. بخش مرتب شده در ابتدای کار طولی برابر با ۱ دارد و که جایگاه آن در خانه اول خواهد بود. سپس ما روی بقیه آرایه که بخش مرتب نشده است، حرکت می کنیم و عنصر مناسب را پیدا و در آن جایگاه قرار می دهیم. با تکرار رویه جستجو در بخش مرتب نشده، هر بار یک عنصر مناسب دیگر را پیدا می کنیم و در بخش مرتب شده قرار می دهیم که با این کار طول بخش مرتب یکی بیشتر می شود.

عنصر مناسبی که در هر بار جستجو تا انتهای بخش دوم (بخش مرتب نشده سمت راست آرایه) پیدا می شود، به بخش مرتب شده اضافه می شود. نحوه اضافه کردن هم به صورت هُل دادن (shift) عناصر به سمت راست است تا جایگاه عنصر خالی شود.

پیاده سازی:

void insertionSort(int *arr*[], int *n*)

{

    int i, key, j;

    for (i = 1; i < *n*; i++)

    {

        key = *arr*[i];

        j = i - 1;

        /\* Move elements of arr[0..i-1], that are

        greater than key, to one position ahead

        of their current position \*/

        while (j >= 0 && *arr*[j] > key)

        {

*arr*[j + 1] = *arr*[j];

            j = j - 1;

        }

*arr*[j + 1] = key;

    }

}

**همانطور که می بینید، حلقه اجرای حلقه مرتب سازی روی آرایه از عنصر دوم شروع شده است، به طور پیشفرض فکر می کنیم عنصر اول در جایگاه مرتب است و این همان ایده تقسیم آرایه به دو بخش مرتب و غیرمرتب است که از ابتدا گفتیم.**

**سپس اولین عنصر از بخش غیرمرتب (همان عنصر دوم آرایه) با آخرین عنصر بخش مرتب شده مقایسه می شود. عنصر غیرمرتب فعلی در متغیر Key به طور موقت نگهداری می شود و اگر بزرگترین عنصر داخل بخش مرتب از مقدار Key هم بزرگتر باشد، آن عنصر و همه سمت راستی هایش هُل داده می شوند تا جای عنصر Key باز شود و به بخش مرتب شده منتقل شود.**

**توجه داشته باشید که این هُل دادن به این صورت است که هر عنصر به جای سمت راستی خود می نشید و در آن کپی می شود. که این کار درون حلقه داخلی while انجام شده است.**

**اما اگر عنصر Key از بالاترین و بزرگترین عنصر بخش مرتب کوچکتر نباشد چه؟ در آن صورت جا به جایی نداریم و عنصر داخل Key به سادگی به انتهای بخش مرتب اضافه می شود و به این شکل به بخش مرتب وارد می شود.**

ویژگی های مرتب سازی درجی چیست؟

**موارد مرزی (Boundary Cases): مرتب سازی درجی، در صورتی که عناصر دارای ترتیب برعکس باشند، بیشترین زمان اجرا را می‌برد. همچنین، در صورتی که عناصر مرتب شده باشند، کمترین زمان اجرا (از درجه n) را خواهد داشت.**

**مرتب‌سازی درجا: بله**

**پایدار: بله**

**آنلاین: بله**

**کاربردها: مرتب‌سازی درجی، هنگامی مورد استفاده قرار می‌گیرد که تعداد عناصر کم باشد. همچنین، هنگامی که آرایه ورودی تقریبا مرتب شده باشد و تنها چند عنصر در یک آرایه بزرگ در جای نادرست قرار داشته باشند، مرتب سازی درجی گزینه مناسبی خواهد بود.**

پیچیدگی زمانی مرتب سازی درجی:

**در ادامه و برای بررسی پیچیدگی زمانی الگوریتم مرتب سازی درجی، به بدترین حالت ممکن در اجرا می پردازیم. اگر در هر مرحله ما مجبور باشیم هر عنصر را در طول کل آرایه و به اندازه آن منتقل کنیم، درجه زمانی آن خواهد بود. حالا این کار را برای تمام عناصر هم انجام دهیم که این یعنی به درجه زمانی می رسیم.**

**جالب است بدانید که این بدترین حالت هم زمانی اتفاق می افتد که آرایه ای که برای مرتب سازی به الگوریتم داده می شود، از قبل به صورت معکوس مرتب باشد؛ مثلاً آرایه نزولی را برای مرتب سازی صعودی به الگوریتم بدهیم.**

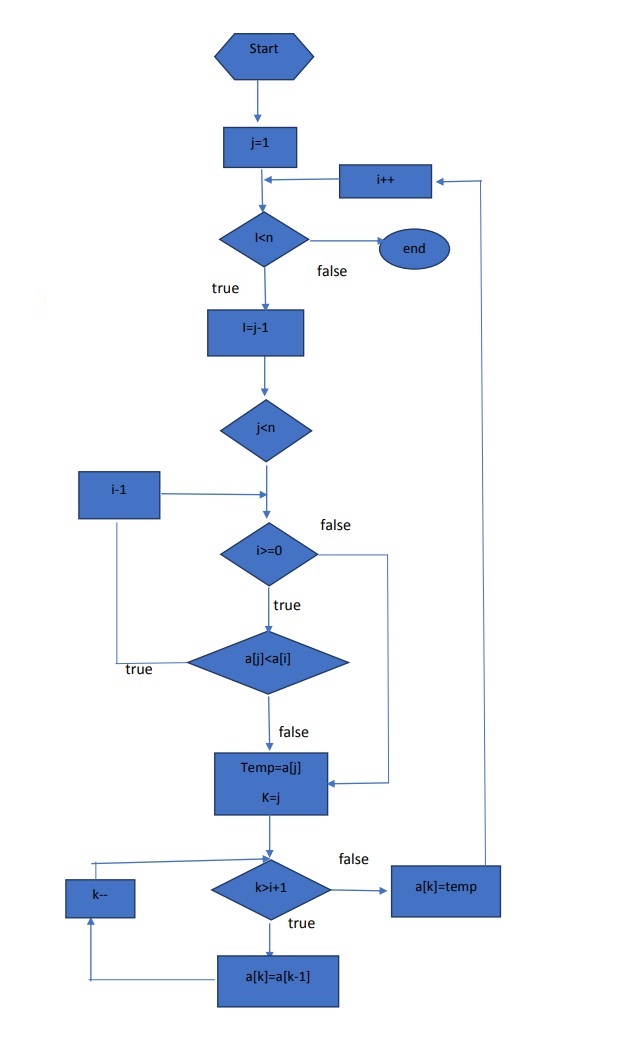
**مقایسه با دیگر الگوریتم های مرتب سازی:**

مرتب‌سازی درجی بسیار شبیه به مرتب‌سازی انتخابی است. مانند مرتب‌سازی انتخابی، پس از k مرحله در آرایه، k عنصر اول در حالت مرتب شده قرار دارند. برای مرتب‌سازی انتخابی، آن عناصر، کوچترین k عنصر موجود در لیست هستند در حالیکه در مرتب‌سازی درجی، آن عناصرk عنصر اول از آرایه مرتب نشده هستند. مزیت مرتب‌سازی درجی این است که برای تشخیص مکان k+1 ام ، فقط عناصر مورد نیاز را بررسی می‌کند؛ درحالیکه مرتب‌سازی انتخابی باید همه عناصر باقی‌مانده را بررسی کند تا کوچکترین آن‌ها را پیدا کند. محاسبات نشان می‌دهد که مرتب‌سازی درجی معمولاً نصف تعداد مقایسه‌های مرتب‌سازی انتخابی را انجام می‌دهد. فرض کنید که اولویت عنصر k+1 ام تصادفی است. حال مرتب‌سازی درجی به‌طور متوسط باید نیمی از k عنصر مرتب شده را بررسی کند تا محل عنصر جدید را پیدا کند، درحالیکه مرتب‌سازی انتخابی همیشه نیازمند بررسی همه عناصر مرتب نشده است.

اگر آرایه ورودی به‌طور معکوس مرتب شده باشد، مرتب‌سازی درجی به اندازه مرتب‌سازی انتخابی مقایسه انجام می‌دهد. ما اگر آرایه ورودی واقعاً مرتب شده است، مرتب‌سازی درجی n-1 مقایسه کمتر انجام می‌دهد، بنابراین وقتی آرایه ورودی «تقریباً مرتب شده» باشد، مرتب‌سازی درجی بهینه تر عمل می‌کند. درحالیکه مرتب‌سازی درجی معمولاً تعداد مقایسه‌های کمتری از مرتب‌سازی انتخابی انجام می‌دهد، نیازمند نوشتن‌های بیشتر است چون حلقه داخلی ممکن است به جابجا کردن بخش‌های زیادی از بخش مرتب شده نیاز داشته باشد.

در حالت کلی، مرتب‌سازی درجی در آرایه بار عمل نوشتن انجام می‌دهد؛ درحالیکه مرتب‌سازی انتخابی تنها بار می‌نویسد. به همین دلیل مرتب‌سازی انتخابی در مواردی که نوشتن در حافظه نیازمند هزینه زیادی باشد، سریعتر عمل می‌کند مانند نوشتن در EEPROM یا حافظه فلش. برخی الگوریتم‌های تقسیم و حل مثل مرتب‌سازی سریع یا مرتب‌سازی ادغام با تقسیم کردن لیست به صورت بازگشتی به زیر لیست‌های مرتب شده، عمل مرتب‌سازی را انجام می‌دهند. در عمل یک راه بهینه‌سازی برای این الگوریتم‌ها این است که مرتب‌سازی درجی را برای مرتب کردن زیر لیست‌های کوچک استفاده کنیم که در کل موجب سریعتر شدن عملیات می‌شود. سایز لیستی که برای آن، مرتب‌سازی درجی مزیت بیشتری از سایر انواع مرتب‌سازی‌ها دارد، با توجه به پیاده‌سازی و محیط تغییر می‌کند اما معمولاً بین هشت تا بیست عنصر است.

**فلوچارت الگوریتم مرتب سازی درجی:**



**بهینه سازی های این الگوریتم:**

آقای D.L. Shell این الگوریتم را بهبود بخشید که نسخه بهبود داده شده آن، مرتب‌سازی شل نامیده می‌شود. این الگوریتم مرتب‌سازی، عناصر را با فاصله‌ای که در هر مرحله کم می‌شود، مقایسه می‌کند. مرتب‌سازی شل به‌طور قابل توجهی مراحل اجرا را در کار عملی ارتقا داده است که در دو نسخه مختلف، با زمان اجرای

و عمل می کند. اگر هزینه مقایسه‌ها از هزینه جابجایی بیشتر باشد، مانند حالتی که لیست ورودی مرتب شده باشد، استفاده کردن از مرتب‌سازی درجی دودویی عملکرد بهتری خواهد داشت. مرتب‌سازی درجی دودویی از جستجوی دودویی استفاده می‌کند تا محل مناسب برای قرار دادن عنصر جدید را پیدا کند و بنابراین در بدترین حالت مقایسه انجام می‌دهد که است.

الگوریتم در حالت میانگین بخاطر مجموعه جابجایی‌هایی که برای هر درج لازم است، زمان اجرای دارد. در بهترین حالت نیز زمان اجرای آن طولانی تر نیست. برای پیشگیری کردن از مجموعه‌ای از جابجایی‌ها برای هر درج، ورودی می‌تواند در یک لیست پیوندی مرتب شود که امکان می‌دهد عناصر در زمان ثابت درج یا حذف شوند. البته اجرای جستجوی دودویی در یک لیست پیوندی غیرممکن است چون یک لیست پیوندی دسترسی تصادفی به عناصر را پشتیبانی نمی‌کند؛ بنابراین زمان اجرا برای جستجو است.

*اگر یک ساختمان داده‌های پیچیده تر مانند هرم استفاده شود، زمان مورد استفاده برای جستجو و درج به‌طور قابل ملاحظه‌ای کاهش می‌یابد.*

***شبیه کد الگوریتم مرتب سازی شل:***

***استفاده از دنباله فاصله‌های Marcin Ciura با یک مرتب‌سازی درجی داخلی***

*# Sort an array a[0...n-1].*

gaps = [701, 301, 132, 57, 23, 10, 4, 1]

*# Start with the largest gap and work down to a gap of 1*

**foreach** (gap in gaps)

{

*# Do a gapped insertion sort for this gap size.*

*# The first gap elements a[0..gap-1] are already in gapped order*

*# keep adding one more element until the entire array is gap sorted*

**for** (i = gap; i < n; i += 1)

{

*# add a[i] to the elements that have been gap sorted*

*# save a[i] in temp and make a hole at position i*

temp = a[i]

*# shift earlier gap-sorted elements up until the correct location for a[i] is found*

**for** (j = i; j >= gap and a[j - gap] > temp; j -= gap)

{

a[j] = a[j - gap]

}

*# put temp (the original a[i]) in its correct location*

a[j] = temp

}

}

در نتیجه:

* الگوریتم مرتب‌سازی درجی درصورتی کارآمد است که داده‌ها تقریباً مرتب باشند.
* مرتب‌سازی درجی معمولاً کم بازده‌است چون مقادیر را در هر زمان فقط به اندازه یک موقعیت جابجا می‌کند.

***محاسبه ی پیچیدگی زمانی:***

*بهترین حالت: زمانی که آرایه مرتب شده باشد.*

*بدترین حالت: زمانی که آرایه به صورت معکوس مرتب شده باشد.*

***مرتب سازی شمارشی:***

*مرتب‌ساز شمارشی یکی از الگوریتم‌های مرتب‌سازی است که (مثل مرتب‌ساز سطلی) با فرض دانستن بازه اعداد داخل آرایه (A)، عمل مرتب‌سازی را انجام می‌دهد. این الگوریتم از این بازه برای ساختن یک آرایه (C) با این طول استفاده می‌کند. هر اندیس i در آرایه C برای شمارش تعداد عناصر A که دارای مقدار i هستند، به کار می‌رود. این اعداد در C برای قرار دادن عناصر A در جای درستشان در آرایه خروجی، به کار می‌روند. مرتب‌ساز لانه‌کبوتری از این الگوریتم، کارآمدتر است.*

***ویژگی های مرتب سازی شمارشی:***

*مرتب‌ساز شمارشی، یک مرتب‌ساز پایدار است و دارای زمان اجرای O است که n و k به ترتیب، طول‌های آرایه‌های A (آرایه ورودی) و C (آرایه شمارشی) هستند. برای این که الگوریتم، کارآمد باشد، k نباید خیلی بیشتر از n باشد.*

*اندیس‌های C، باید از کوچکترین تا بزرگترین عناصر A باشند تا بتوان C را به صورت مستقیم با مقادیر A، اندیس‌گذاری کرد. در غیر این صورت، مقادیر A باید انتقال (شیفت) داده شوند تا کمترین مقدار A، معادل کوچکترین اندیس C شود. اگر بیشتری و کمترین مقادیر A معلوم نباشند، باید توسط یک الگوریتم انتخاب، که زمان می‌گیرد، آنها را پیدار کرد. طول آرایه شمارشی C، حداقل باید برابر بازه اعداد ورودی باشد. (کمترین منهای بیشتری و به‌اضافه ۱). این ویژگی باعث می‌شود که استفاده از مرتب‌ساز شمارشی برای بازه‌های بزرگ اعداد، غیرعملی شود. مرتب‌ساز شمارشی، برای مثال، می‌تواند بهترین الگوریتم برای اعدادی باشد که بین ۰ و ۱۰۰ قرار دارند. این الگوریتم برای مرتب کردن اسامی بر اساس حروف الفبا، نامناسب است.*

*به علت اینکه مرتب‌ساز شمارشی، از مقادیر به عنوان اندیس آرایه استفاده می‌کند، یک الگوریتم مرتب‌ساز مقایسه‌ای، نیست و بنابراین کران پایین (Ω(n log n برای این الگوریتم قابل تطبیق نیست.*

***الگوریتم:***

1. *شروع*
2. *کوچکترین و بزرگترین عناصر مجموعه را پیدا کن.*
3. *مقادیر مختلف موجود در آرایه را بشمار. (برای مثال، مجموعه [۴،۴،۴،۱،۱] دارای سه تا ۴ و دو تا ۱ است)*
4. *شمارش‌ها را جمع کن.*
5. *آرایه مقصد را از انتها پر کن: هر عنصر را در موقعیت countام قرار بده.*
6. *هر موقع که عنصری را درج می‌کنی، شمارشش را یکی کم کن.*
7. *پایان*

***پیاده سازی:***

void counting\_sort(int *A*[], int *n*)

{

    int i;

    int Max = *A*[0];

    int \*B;

    int \*C;

    //found Max number in array

    for (i = 1; i < *n*; i++)

    {

        if (Max < *A*[i])

        {

            Max = *A*[i];

        }

    }

    //Define array that length is Max

    B = **new** int[Max + 1];

    //set 0 all array

    for (i = 0; i <= Max; i++)

    {

        B[i] = 0;

    }

    //

    for (i = 0; i < *n*; i++)

    {

        B[*A*[i]]++;

    }

    //

    for (i = 1; i <= Max; i++)

    {

        B[i] = B[i] + B[i - 1];

    }

    //make new array for insert sorted number in it

    C = **new** int[*n*];

    //

    for (i = *n* - 1; i >= 0; i--)

    {

        C[B[*A*[i]] - 1] = *A*[i];

        B[*A*[i]]--;

    }

    //put C in A and delete C from memory

    for (i = 0; i < *n*; i++)

    {

*A*[i] = C[i];

    }

}

***پیچیدگی زمانی این الگوریتم:***

|  |  |
| --- | --- |
| **for-loop** | **time of counting** |
| 1st | O(max) |
| 2nd | O(size) |
| 3rd | O(max) |
| 4th | O(size) |

*بدترین حالت:*

*بهترین حالت:*

*حالت متوسط:*

تمامی حالت ها باهم مساوی هستند ، زیرا مهم نیست آرایه به چه صورتی چیده شده باشد ، این الگوریتم از n+k بار آرایه را بررسی می کند.

**مرتب سازی هرمی:**

هیپ یک ساختمان داده خاص مبتنی بر درخت است و از این رو هیپ را گره‌هایی تشکیل می‌دهند. ما عناصر هیپ را به این گره‌ها انتساب می‌دهیم. هر گره شامل دقیقاً یک عنصر است. ضمناً گره‌ها می‌توانند فرزندانی داشته باشند. اگر یک گره هیچ فرزندی نداشته باشد آن را برگ می‌نامیم. آنچه هیپ را خاص می‌سازد دو چیز است:

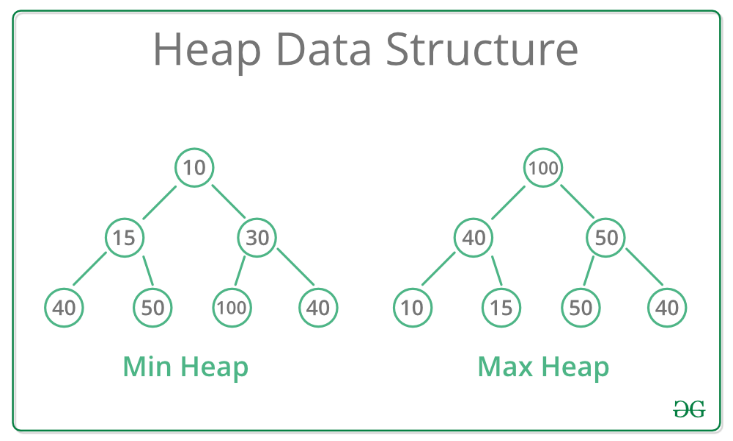
1. مقدار هر گره باید کمتر یا مساوی مقادیر ذخیره شده در فرزندان آن باشد.
2. هیپ یک درخت کامل است یعنی کمترین ارتفاع ممکن را دارد.

به دلیل قاعده اول فوق کمترین عنصر همواره در ریشه درخت قرار می‌گیرد. روش الزام این قواعد نیز به نوع پیاده‌سازی وابسته است. هیپ‌ها عموماً برای پیاده‌سازی صف‌های اولویت استفاده می‌شوند، زیرا هیپ یک پیاده‌سازی کاملاً کارآمد برای استخراج عنصر با کمترین (یا بیشترین) مقدار محسوب می‌شود.

**انواع هیپ**

هیپ گونه‌های بسیار مختلفی دارد که تنها تفاوت آن‌ها از نظر برخی جزییات پیاده‌سازی با هم متفاوت هستند. برای نمونه آن چه در بخش فوق توصیف کردیم، یک Min-Heap یا هرم کمینه است، زیرا مقدار والد همواره کمتر از فرزندانش است. به طور جایگزین می‌توان Max-Heap یا هرم بیشینه نیز داشت که در آن والد همواره بزرگ‌تر از فرزندانش است. از این رو بزرگ‌ترین عنصر در گره ریشه قرار خواهد داشت.

ما می‌توانیم از میان پیاده‌سازی‌های مختلف درخت یکی را برای هیپ انتخاب کنیم. سرراست‌ترین گزینه درخت دودویی است. در درخت دودویی هر گره می‌تواند حداکثر دو فرزند داشته باشد. ما آن‌ها را برگ چپ و برگ راست می‌نامیم. ساده‌ترین روش برای الزام به قاعده دوم بخش فوق استفاده از درخت دودویی کامل است. یک درخت دودویی کامل داری قواعد ساده‌ای به شرح زیر است:

* اگر یک گره تنها یک فرزند داشته باشد، این گره باید برگ چپ باشد.
* تنها گره سمت راست روی عمیق‌ترین سطح می‌تواند دقیقاً یک فرزند داشته باشد.
* برگ‌ها می‌توانند صرفاً در عمیق‌ترین سطح باشند.

**درج عناصر**

ما باید همه عملیات را به ترتیبی پیاده‌سازی کنیم که هیپ بدون تغییر بماند. بدین ترتیب می‌توانیم هیپ را با استفاده از درج‌های مکرر بسازیم. بنابراین در ادامه روی یک عمل درج منفرد متمرکز می‌شویم:

1. یک برگ جدید بسازید که سمت راست‌ترین جایگاه ممکن روی عمیق‌ترین سطح است و آیتم را در این گره ذخیره کنید.
2. اگر این عنصر کمتر از والدینش باشد، جای آن‌ها را با هم عوض می‌کنیم.
3. گام 2 را تا زمانی که عنصر کمتر از والدینش باشد و یا به ریشه جدید تبدیل شود ادامه می‌دهیم.

توجه کنید که گام 2 فوق، قاعده هیپ را نقض نمی‌کند، زیرا اگر مقدار یک گره را با مقدار کمتر عوض کنیم همچنان کمتر از فرزندانش خواهد بود.

**انواع هرم**

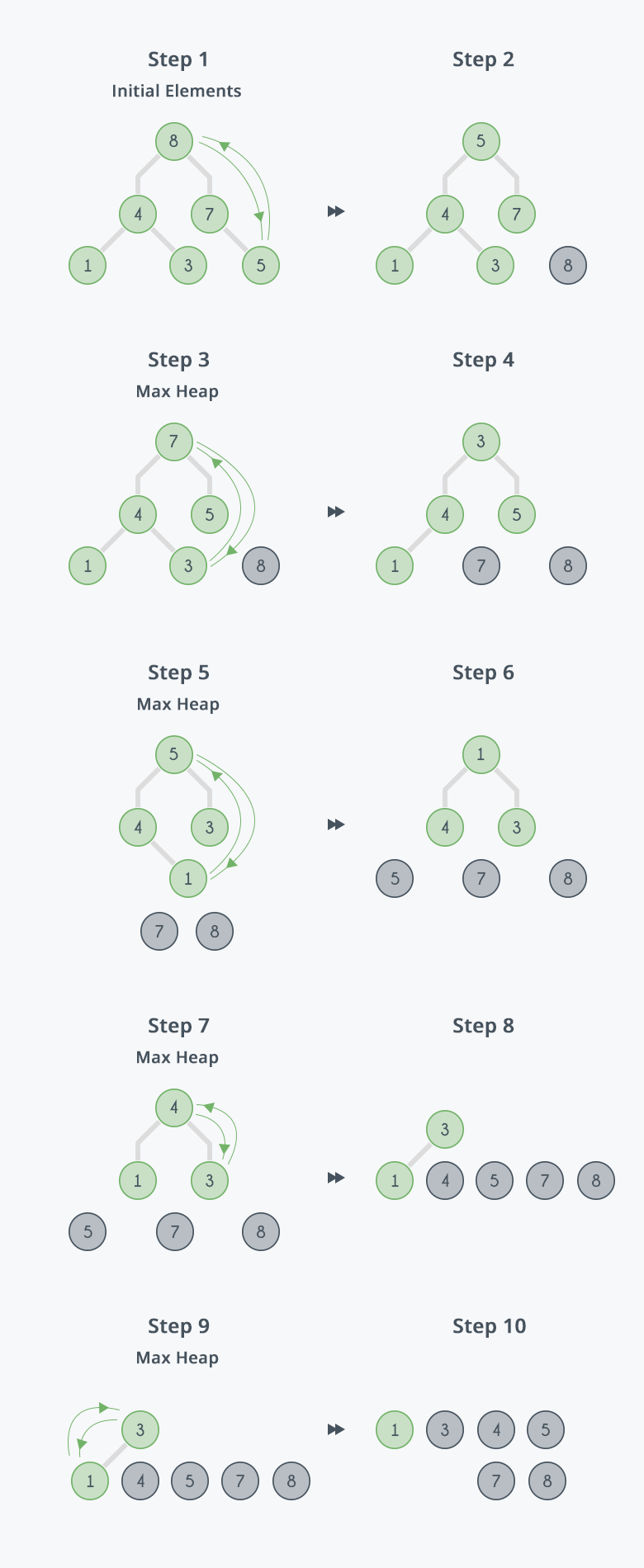
* هرم ۲–۳
* بیپ ‏(Beap)‏
* هرم دودویی
* هرم فیبوناچی
* هرم چپ‌گرا
* هرم اریب
* هرم بی
* هرم دو جمله ای
* هرم مبنایی

کاربرد ها

**ساختار داده هرم کاربردهای زیادی دارد که برخی از آنها عبارتند از:**

* **مرتب‌سازی هرمی: یکی از بهترین شیوه‌های مرتب‌سازی است که «درجا» (**in place**) انجام می‌شود و در بدترین حالت نیز مرتبه زمانی خطی دارد.**
* **الگوریتم انتخاب: برای یافتن عنصر بیشینه، کمینه، میانه و** k**امین عنصر در زمان خطی به کار می‌رود.**
* **الگوریتم‌های گراف: با استفاده از هرم به عنوان زمان بهینه پیمایش، زمان اجرا به صورت چند جمله‌ای کاهش می‌یابد. این الگوریتم‌ها برای مواردی مانند الگوریتم پریم و الگوریتم دیکسترا استفاده می‌شوند.**
* **صف اولویت دار: برای پیاده‌سازی صف اولویت دار معمولاً از هرم کمینه استفاده می‌شود.**

نحوه ی مرتب سازی هرمی:



توضیح مرتب سازی:

**مرتب‌سازی هرمی نوعی الگوریتم است که در آن از مقایسه برای چینش یک آرایه یا فهرست استفاده می‌شود. این الگوریتم بخشی از خانواده مرتب‌سازی انتخابی است. با وجود اینکه در اکثر رایانه‌ها از الگوریتم چینش سریع کندتر است ولی در بدترین حالت سرعت بالاتر را دارا می‌باشد. این الگوریتم در محل است، ولی حالت پایداری ندارد.**

**در این مرتب‌سازی، ابتدا از کل آرایه داده شده یک درخت مکس هیپ (یا درخت مین هیپ) می‌سازد. سپس بزرگترین مقدار را برمی‌دارد و در انتهای آرایه مرتب شده قرار می‌دهد. بعد از حذف بزرگترین مقدار، دوباره از بقیه اعداد یک درخت مکس هیپ می‌سازد تا دومین عدد بزرگ یافت شود. بزرگ‌ترین مقدار در بین مقادیر باقی‌مانده را برمی‌دارد و آن را در مکان یکی قبل از انتهای آرایه قرار می‌دهد. این کار تا زمانی که هیچ مقداری در هرم باقی نماند و آرایه مرتب شده کامل شود، تکرار می‌شود.**

**یکی از روش‌های مرتب‌سازی داده‌هااست، که براساس خصوصیات درخت** heap **پیاده‌سازی شده‌است. بر اساس تعریف درخت** heap**، در یک** max-heap **یا** min-heap **بزرگترین یا کوچکترین مقدار بین داده‌ها همواره در ریشه درخت قرار دارد. یافتن بزرگترین یا کوچکترین عنصر بین عناصر، هزینه ثابت دارد. با حذف این عنصر از درخت، بزرگترین یا کوچکترین عنصر بعدی مجدداً در ریشه قرار می‌گیرد. به این ترتیب با حذف متوالی عناصر درخت** heap **و درج آن‌ها در محل جدید، یک آرایه مرتب‌شده نزولی ویا صعودی به دست خواهد آمد.**

بررسی پیچیدگی زمانی مرتب سازی هرمی

**برای مرتب‌کردن عناصر با استفاده از درخت** heap**، نیاز به** n **عمل حذف گره ریشه از درخت دارد. عمل حذف گره ریشه در درخت** heap **خود از مرتبه**  است. در نتیجه کل این عملیات از مرتبه **خواهد بود. نکته قابل توجه دیگر، ساخت درخت** heap **از عناصر مورد نظر برای مرتب‌سازی است. در حالت عادی عناصر به صورت نامرتب و با چیدمان تصادفی در اختیار هر الگوریتم مرتب‌سازی قرار می‌گیرند. در این حالت یک هزینه زمانی دیگر برای ساخت درخت** heap **از روی چنین فهرستی مورد نیاز است.**

پیاده سازی مرتب سازی هرمی:

void max\_heapify(int \**arr*, int *n*, int *i*)

{

    int left\_child = 2 \* *i* + 1;

    int right\_child = 2 \* *i* + 2;

    int large = *i*;

    if (left\_child < *n* && *arr*[large] < *arr*[left\_child])

    {

        large = left\_child;

    }

    if (right\_child < *n* && *arr*[large] < *arr*[right\_child])

    {

        large = right\_child;

    }

    if (large != *i*)

    {

        int temp;

        temp = *arr*[*i*];

*arr*[*i*] = *arr*[large];

*arr*[large] = temp;

        max\_heapify(*arr*, *n*, large);

    }

}

void make\_max\_heap(int \**arr*, int *n*)

{

    for (int i = *n* / 2 + 1; i >= 0; i--)

    {

        max\_heapify(*arr*, *n*, i);

    }

}

void heap\_sort(int \**arr*, int *n*)

{

    make\_max\_heap(*arr*, *n*);

    int temp;

    for (int i = *n* - 1; i > 0; i--)

    {

        temp = *arr*[0];

*arr*[0] = *arr*[i];

*arr*[i] = temp;

        max\_heapify(*arr*, i, 0);

    }

}

*مرتب‌سازی انتخابی*

*مرتب‌سازی انتخابی یکی از انواع الگوریتم مرتب‌سازی می‌باشد که جزو دسته الگوریتم‌های مرتب‌سازی مبتنی بر مقایسه‌است. این الگوریتم دارای پیچیدگی زمانی از درجه است که به همین دلیل اعمال آن روی مجموعهٔ بزرگی از اعداد کارا به نظر نمی‌رسد وبطورعمومی ضعیف تر ازنوع مشابهش که مرتب‌ساز درجی است عمل می‌کند. این مرتب‌سازی به دلیل سادگی اش قابل توجه‌است. کارایی آن برحسب تعداد ورودی‌ها در نمودار زیر نشان داده شده‌است.*

***نحوه عملکرد***

*این الگوریتم این‌گونه عمل می‌کند: ابتدا کوچکترین عنصر مجموعه اعداد را یافته با اولین عدد جابجا می‌کنیم. سپس دومین عنصر کوچکتر را یافته با دومین عدد جابجا می‌کنیم و این روند را برای n-1 عدد اول تکرار می‌کنیم. در حقیقت در هر مرحله ما لیست خود را به دو بخش تقسیم می‌کنیم. زیرلیست اول که قبلاً مرتب کرده‌ایم و سایر اعضای لیست که هنوز مرتب نشده‌است.*

***پیاده سازی***

*الگوریتم مرتب سازی انتخابی دو زیر آرایه از آرایه داده شده ایجاده می کند:*

* *زیر آرایه ای که مقادیر مرتب شده در آن قرار می گیرند.*
* *زیر آرایه ای که مقادیر نامرتب در آن قرار دارند.*

*در هر مرحله از تکرار حلقه درون الگوریتم، ما به عنوان مرجع مقایسه اولین عنصر بخش مرتب نشده را به عنوان کوچکترین فرض کرده و سپس در بخش مرتب نشده جلو رفته ایم و با مقایسه های متوالی سعی کرده ایم کوچکترین عنصر این بخش را پیدا کنیم.*

*هنگامی که ما کوچکترین عنصر بخش مرتب نشده فعلی را یافتیم، آن عنصر را با اولین عنصر بخش مرتب نشده (که بلافاصله بعد از اعضای مرتب شده می آید) جا به جا می کنیم.*

*بنابراین آن کوچکترین عنصر بخش مرتب نشده به انتهای بخش مرتب شده اضافه می شود و با این کار طول بخش مرتب شده یکی اضافه می شود و از طول بخش مرتب نشده کاسته می شود.*

*با تکرار این کار به اندازه تعداد عناصر، آرایه مرتب شده به دست می آید.*

لیست اعداد زیر را در نظر بگیرید که باید به صورت صعودی(کوچک به بزرگ)مرتب شود:

2 8 4 1 7

در مرحله‌ی اول، کل لیست از ابتدا تا انتها بررسی شده و بزرگترین عنصر با عنصر انتهای لیست نامرتب جابجا می‌شود:

1)    2 8 4 1 7    →    2 7 4 1 **8**

در مرحله‌ی دوم، پیمایش از ابتدای لیست تا عنصر چهارم صورت گرفته و بزرگترین عنصر با عنصر انتهای آن جابجا می‌شود:

2)    2 7 4 1 **8**    →    2 1 4 **7 8**

علت این که چرا عنصر پنجم بررسی نمی‌شود کاملا مشخص است. این عنصر در مرحله‌ی قبل به عنوان بزرگترین عنصر به انتهای لیست منتقل شده است و به طور حتم نیاز به جابجایی ندارد.  
در مرحله‌ی سوم، عناصر اول تا سوم بررسی شده و بزرگترین عنصر به انتهای آن منتقل می‌شود:

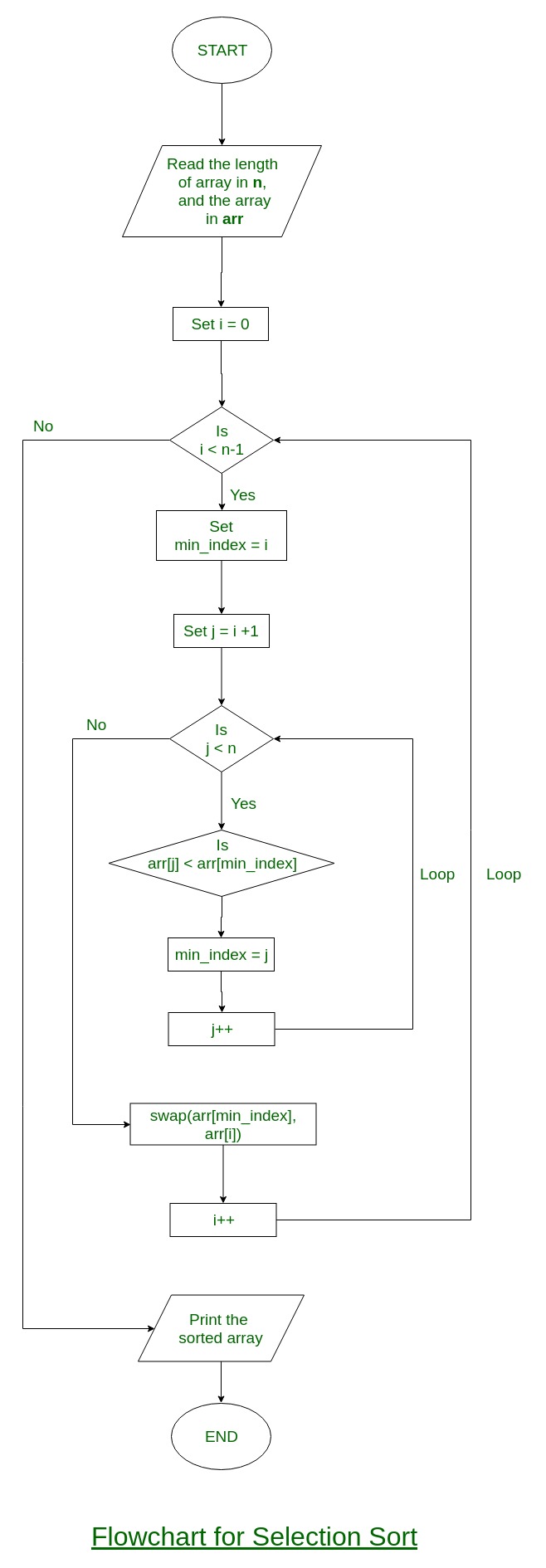
3)    2 1 4 **7 8**    →    2 1 **4 7 8**

و در مرحله‌ی آخر دو عنصر باقیمانده مقایسه می‌شوند:

4)    2 1 **4 7 8**    →    **1 2 4 7 8**

و به این ترتیب لیست مرتب می‌شود.

***فلوچارت مرتب سازی انتخابی:***



***کد پیاده سازی شده مرتب سازی انتخابی:***

int i, j, max, temp; // calculate

  for (i = n - 1; i > 0; i--)

  {

    max = 0;

    for (j = 1; j <= i; j++)

    {

      if (arr[max] < arr[j])

      {

        max = j;

      }

    }

    temp = arr[i];

    arr[i] = arr[max];

    arr[max] = temp;

  }

***تحلیل مرتبه الگوریتم***

*تحلیل الگوریتم مرتب‌سازی انتخابی برخلاف بسیاری از مرتب‌سازی‌های دیگر بسیار ساده‌است. زیرا که هیچ‌کدام از حلقه‌های آن به اعداد موجود در لیست ورودی بستگی ندارد. در مرحله اول به دست آوردن کمینه در لیست n عنصری نیاز به پیمودن کل n عدد و n – 1 مقایسه دارد و سپس باید کمینه بدست آمده با اولین عدد جابجا شود. در مرحله بعدی به دست آوردن دومین کمینه در لیست 1 - n عنصری نیاز به پیمودن کل 1 - n عدد و 2 - n مقایسه دارد و کمینه بدست آمده بادومین عدد جابجا شودو این روند ادامه پیدا می‌کند. پس کلاً تعداد مقایسه‌ها عبارتست از:*

مرتبه این الگوریتم به دلیل عدم وابستگی آن به نحوه ترتیب اعداد در بهترین، بدترین و حالت متوسط یکسان و برابر است.

***ویژگی‌های مرتب‌سازی انتخابی***

1. با توجه به قطعه کد نوشته شده، ترتیب عناصر تغییری در عملکرد آن اینجا نمی‌کند. یعنی این الگوریتم برای داده‌های کاملاً مرتب، نامرتب تصادفی و مرتب معکوس به یک ترتیب عمل کرده و تمام مقایسه‌های محاسبه شده در رابطه فوق را انجام می‌دهد؛ بنابراین پیچیدگی این الگوریتم در بهترین حالت و حالت متوسط نیز است.
2. مرتب‌سازی انتخابی یک روش مرتب‌سازی درجا است. یعنی عملیات مرتب‌سازی در داخل خود لیست و بدون نیاز به حافظه کمکی بزرگ انجام می‌گیرد.
3. در پیاده‌سازی مرتب‌سازی انتخابی به روش فوق، اگر دو عنصر با مقدار بیشینه داشته باشیم، اولی انتخاب شده و به انتهای لیست منتقل می‌شود. در نتیجه ترتیب آن‌ها به هم می‌خورد؛ بنابراین این پیاده‌سازی روش پایدار نیست. در روش پایدار ترتیب عناصر با مقدار یکسان تغییر نمی‌کند. اما اگر در مقایسه عناصر آرایه به جای > از => استفاده کنید، مرتب‌سازی پایدار خواهد شد.

***مقایسه با سایر الگوریتمهای مرتب‌سازی***

*این الگوریتم بین الگوریتم‌های با مرتبه مرتب‌ساز انتخابی از مرتب‌ساز حبابی و مرتب‌ساز گورزاد (در حالت متوسط) بهتر عمل می‌کند.*

*اما عموماً ضعیفتر ازمرتب‌ساز درجی است. یکی از شباهتهای مرتب‌ساز درجی به مرتب‌ساز انتخابی این است که در هر دو پس از پیمایش Kام مجموعه اعداد kعنصر اول در جای صحیح خود قرار گرفته‌اند. فایده مرتب‌ساز درجی این است که برای پیدا کردن عنصر x هر تعداد از اعداد را که نیاز است بررسی می‌کند.*

*حال آنکه مرتب ساز انتخابی همه عناصر باقی‌مانده را بررسی می‌کند. به هر حال هر دو آن‌ها روی لیستهای کوچک بسیار سریع هستند. در نهایت اعمال مرتب‌ساز انتخابی روی لیستهایی با اندازه بزرگ به مراتب از مرتب‌ساز حبابی که روی ان لیستها با پیچیدگی زمانی عمل می‌کند ضعیف‌تر است.*

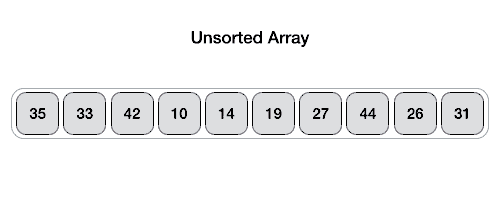
***مرتب سازی سریع***

*مرتب‌سازی سریع یک الگوریتم مرتب‌سازی بسیار بهینه است که بر اساس افراز یک آرایه به آرایه‌های کوچک‌تر عمل می‌کند. یک آرایه بزرگ به دو آرایه کوچک‌تر افراز می‌شود که یکی از آن‌ها مقادیری کوچک‌تر از یک مقدار مشخص را نگه‌داری می‌کند، فرض کنید این مقدار pivot نام دارد. بر همین اساس آرایه دیگر مقادیر بزرگ‌تر از Pivot را نگه‌داری می‌کند.*

*الگوریتم مرتب‌سازی سریع یک آرایه را افراز می‌کند و سپس خود را به صورت بازگشتی مجدداً فراخوانی می‌کند تا دو آرایه فرعی حاصل را مرتب‌سازی کند. این الگوریتم برای مجموعه داده‌های بزرگ کاملاً بهینه است، چون پیچیدگی سناریوی حالت متوسط برابر با و پیچیدگی زمانی بدترین حالت برابر با است که n تعداد آیتم‌ها است.*

***افراز در مرتب‌سازی سریع***

*در تصویر متحرک زیر روش یافتن مقدار pivot در آرایه را می‌بینید:*



*مقدار pivot لیست را به دو بخش تقسیم می‌کند و سپس به طور بازگشتی مقدار pivot را برای هر لیست فرعی پیدا می‌کنیم تا جایی که لیست‌ها تنها حاوی یک عنصر باشند.*

***الگوریتم pivot مرتب‌سازی سریع***

*بر اساس درکی که از افراز در مرتب‌سازی سریع داریم، اینک می‌توانیم الگوریتم آن را به صورت زیر بنویسیم:*

*گام 1- مقدار بالاترین اندیس را به عنوان مقدار pivot انتخاب کن*

*گام 2 – دو متغیر را برای اشاره به سمت چپ و راست لیست به جز pivot انتخاب کن.*

*گام 3 – متغیر چپ به اندیس پایین اشاره می‌کند.*

*گام 4- متغیر راست به اندیس بالا اشاره می‌کند.*

*گام 5- مادمی که مقدار متغیر چپ کمتر از pivot باشد، به سمت راست حرکت کن.*

*گام 6 – مادامی که متغیر راست بزرگ‌تر از pivot باشد، به سمت چپ حرکت کن.*

*گام 7 – اگر هر دو وضعیتی که در گام‌های 5 و 6 اشاره کردیم مطابقت نداشتند، جای متغیرهای راست و چپ را تغییر بده*

*گام 8 – اگر چپ > راست بود در این صورت جایی که این دو به هم می‌رسند، pivot جدید است.*

***الگوریتم مرتب‌سازی سریع***

*با استفاده بازگشتی از الگوریتم pivot در نهایت به کوچک‌ترین اندازه ممکن برای افرازها می‌رسیم. سپس هر افراز برای مرتب‌سازی سریع مورد پردازش قرار می‌گیرد. الگوریتم بازگشتی برای مرتب‌سازی سریع به صورت زیر است:*

*گام 1 – اندیسی که در انتهای سمت راست قرار دارد را pivot قرار بده*

*گام 2 – آرایه را با استفاده از این مقدار pivot افراز کن*

*گام 3 – بخش چپ افراز را با استفاده از مرتب‌سازی سریع به طور بازگشتی مرتب کن.*

*گام 4 – بخش راست افراز را با استفاده از مرتب‌سازی سریع به طور بازگشتی مرتب کن.*

*مثال اجرایی*

0)  3 0 6 1 7 2 5:  low = 0, high = 6

1)  quick\_sort(arr, 0, 6 )    ->    1 0 2 **3** 7 6 5, pivot = 3

2)  quick\_sort(arr, 0, 2 )    ->    0 **1** 2 **3** 7 6 5, pivot = 1

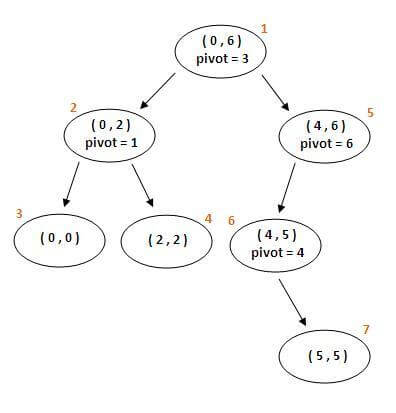
3)  quick\_sort(arr, 0, 0 )    ->    **0** **1** 2 **3** 7 6 5

4)  quick\_sort(arr, 2, 2 )    ->    **0 1 2 3** 7 6 5

5)  quick\_sort(arr, 4, 6 )    ->    **0 1 2 3** 5 6 **7**, pivot = 6

6)  quick\_sort(arr, 4, 5 )    ->    **0 1 2 3 5** 6 **7**, pivot = 4

7)  quick\_sort(arr, 5, 5 )    ->    **0 1 2 3 5 6 7**



***پیاده سازی***

//partition function

int partition(int *A*[], int *p*, int *q*)

{

    int temp;

    int x = *A*[*p*];

    int i = *p*;

    for (int j = *p* + 1; j <= *q*; j++)

    {

        if (*A*[j] <= x)

        {

            i++;

            temp = *A*[j];

*A*[j] = *A*[i];

*A*[i] = temp;

        }

    }

    temp = *A*[i];

*A*[i] = *A*[*p*];

*A*[*p*] = temp;

    return i;

}

//quick sort

void Quick\_sort(int *A*[], int *p*, int *r*)

{

    int q;

    if (*p* < *r*)

    {

        q = partition(*A*, *p*, *r*);

        Quick\_sort(*A*, *p*, q - 1);

        Quick\_sort(*A*, q + 1, *r*);

    }

}

***پیچیدگی زمانی مرتب‌سازی سریع***

*بهترین حالت قرار گرفتن عنصر محوری زمانی است که این عنصر در وسط بازه قرار بگیرد و آن بازه را به دو بازه با اندازه‌ی تقریبا برابر تقسیم کند. در این حالت هر زیربازه مقایسه نیاز خواهد داشت و می‌توان نوشت:*

*با استفاده از قضیه‌ی اصلی یا حل رابطه‌ی بازگشتی مشخص می‌شود که از مرتبه اجرای است.* *در بدترین حالت عنصر محوری در یک سوی بازه قرار گرفته و تمامی عناصر دیگر در یک سمت آن جمع می‌شوند. این حالت زمانی اتفاق می‌افتد که بازه از قبل به صورت نزولی یا صعودی مرتب باشد. در نتیجه یک زیربازه از اندازه‌ی صفر و یک زیربازه از اندازه‌ی n-1 تولید خواهد شد:*

حل این رابطه‌ی بازگشتی نشان از مرتبه‌ی اجرایی در بدترین حالت اجرای الگوریتم دارد.

***ویژگی‌های مرتب‌سازی سریع***

1. *پیچیدگی زمانی اجرای الگوریتم در بهترین حالت و در بدترین حالت است. با استفاده از محاسبات ریاضی می توان نشان داد در حالت متوسط نیز مرتبه ی اجرا است.*
2. *این الگوریتم یک مرتب‌سازی درجا است. یعنی میزان حافظه‌ی مصرفی الگوریتم مستقل از طول آرایه است.*
3. *زمانی که تعداد عناصر آرایه کم باشد، سرعت اجرای مرتب‌سازی درجی بهتر از مرتب‌سازی سریع است. به همین دلیل طی مراحل بازگشتی مرتب‌سازی سریع، اگر طول بازه عدد کوچکی باشد، معمولا بازه با مرتب‌سازی درجی مرتب می‌شود.*
4. *الگوریتم مرتب‌سازی سریع با پیاده‌سازی فوق یک روش ناپایدار است. چنین الگوریتمی لزوما ترتیب عناصر با مقدار یکسان را پس از مرتب‌سازی حفظ نمی‌کند.*
5. *انتخاب عنصر محوری بحث مفصلی دارد. اما در کل یک انتخاب آزاد است. می‌توان عنصر اول، عنصر آخر، یا هر عنصر دیگری را به عنوان عنصر محوری انتخاب کرد. حتی لازم نیست از ابتدا تا انتها از یک روش انتخاب استفاده کرد. یکی از روش‌های رایج، انتخاب یک عنصر تصادفی به عنوان عنصر محوری است. اگرچه انتخاب عنصر محوری مناسب باعث بالا رفتن کارایی الگوریتم می‌شود، اما عموما هزینه‌ی لازم برای یافتن چنین محوری بالا بوده و مقرون به صرفه نیست.*

***پیاده‌سازی صنعتی***

*الگوریتم مرتب‌سازی در دنیای واقعی برای آرایه نسبتاً کوچک مناسب نیست. به علاوه بخش پارتیشن خود نیز مشکل بزرگی در زمان اجرا می‌باشد. برای همین پیشنهاد می‌گردد برای آرایه‌هایی از طول کمتر از ۷ از مرتب‌سازی‌های دیگر مانند مرتب‌سازی درجی یا حبابی استفاده شود.*

*به علاوه به جای پیاده‌سازی بخش partition به صورت عادی با احتمالاتی می‌توان از میانه ۹ برای آرایه‌های بزرگ (بیش از ۴۰ درایه) و میانه ۳ برای ارایه‌های متوسط (کمتر از ۴۰ درایه) و عضو وسط برای آرایه‌های کوچک استفاده کرد.*

*به علاوه در چنین پیاده‌سازی‌هایی ابتدا اعداد صفر (برای آرایه از اعداد مثبت) را ابتدا به شروع آرایه منتقل می‌کنند. و همچنین درایه‌های غیر عددی را نیز هندل می‌کنند تا در اجرای الگوریتم اختلالی به وجود نیاورد.*

برای توضیحات بیشتر درباره نسخه‌های بهینه مرتب‌سازی سریع می‌توانید به مرجع بنتلی و مک ایلوری مراجعه نمایید.

***منابع و ماخذ***

[*https://web.archive.org/web/20100209214318/http://www.miaowang.de/studium/tutorials/applets/selectionsort\_en.html*](https://web.archive.org/web/20100209214318/http://www.miaowang.de/studium/tutorials/applets/selectionsort_en.html)

[*https://blog.faradars.org/sorting-lgorithms/#%D8%A7%D9%84%DA%AF%D9%88%D8%B1%DB%8C%D8%AA%D9%85\_%D9%85%D8%B1%D8%AA%D8%A8%E2%80%8C%D8%B3%D8%A7%D8%B2%DB%8C\_%D8%A7%D9%86%D8%AA%D8%AE%D8%A7%D8%A8%DB%8C\_(Selection\_sort)*](https://blog.faradars.org/sorting-lgorithms/#%D8%A7%D9%84%DA%AF%D9%88%D8%B1%DB%8C%D8%AA%D9%85_%D9%85%D8%B1%D8%AA%D8%A8%E2%80%8C%D8%B3%D8%A7%D8%B2%DB%8C_%D8%A7%D9%86%D8%AA%D8%AE%D8%A7%D8%A8%DB%8C_(Selection_sort))

[*https://fa.wikipedia.org/wiki/%D9%85%D8%B1%D8%AA%D8%A8%E2%80%8C%D8%B3%D8%A7%D8%B2%DB%8C\_%D8%B3%D8%B1%DB%8C%D8%B9*](https://fa.wikipedia.org/wiki/%D9%85%D8%B1%D8%AA%D8%A8%E2%80%8C%D8%B3%D8%A7%D8%B2%DB%8C_%D8%B3%D8%B1%DB%8C%D8%B9)

[*https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/*](https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/)

Dean, B. C. (2006). "A simple expected running time analysis for randomized 'divide and conquer' algorithms". Discrete Applied Mathematics

[*Hoare, C. A. R.*](https://en.wikipedia.org/wiki/Tony_Hoare) (1961). "Algorithm 65: Find". [*Comm. ACM*](https://en.wikipedia.org/wiki/Communications_of_the_ACM).

[Faron Moller](https://en.wikipedia.org/wiki/Faron_Moller). [Analysis of Quicksort](http://www.cs.swan.ac.uk/~csfm/Courses/CS_332/quicksort.pdf). CS 332: Designing Algorithms. Department of Computer Science