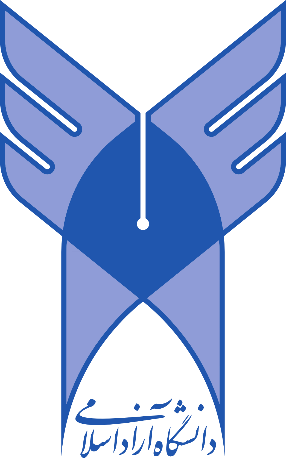
**C:\Program Files (x86)\Hamoon Soft\Chalipa\Chalipa Clipart\Brdr01.wmf**

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 )

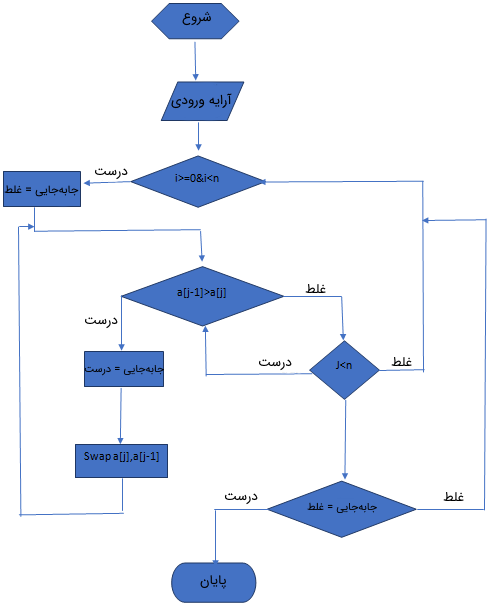
در حال حاضر، آرایه مرتب شده است، اما الگوریتم نمی‌داند که آیا کار به پایان رسیده یا خیر؛ بنابراین، به یک دور کامل دیگر بدون انجام هرگونه جا به جایی نیاز دارد تا بفهمد که مرتب‌سازی با موفقیت به پایان رسیده است.

( 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.

**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.