بسم الله الرّحمن الرّحیم

رامین اجلال

تمرین ۱

۸۸۰۸۷۹۰۰٤

الگوریتمهای مرتب سازی و جستجو

الگوریتم های جستجو

یک الگوریتم جستجو به طور کلی الگوریتمی است که درون یک مجموعه از داده ها که توسط یک نوع ساختمان داده ذخیره شده اند مکان یک مقدار داده شده به عنوان آرگومان جستجو را درون ساختمان داده مشخص می کند، یا تعیین می کند در مجموعه وجود دارد یا خیر.

جستجوی خطی

جستجوی دودوئی

جستجوی خطی

جستجوی خطی (linear search) یا جستجوی ترتیبی (sequential search) کلیه عناصر درون یک لیست را یکی یکی بررسی می کند تا آرگومان جستجو پیدا شود.

شبه کد زیر روش جسجوی خطی را نشان می دهد. مقدار x درون آرایه A با n عنصر جستجو می شود و موقعیت آنرا بر می گرداند. اگر در آرایه وجود نداشته باشد صفر برگردانده می شود.

```
i:=0;
for i:=1 to n do
  if A[i] = x then
    break
  end if
end for
Return i
```

اگر تعداد عناصر مجموعه n باشد، زمان جستجو O(n) است. بهترین حالت زمانی اتفاق می افتد که آرگومان جستجو برابر با اولین عنصر لیست باشد که با یک مقایسه پیدا می شود. بدترین حالت وقتی است که داده درون لیست وجود ندارد یا در انتهای لیست واقع شده است که n مقایسه مورد نیاز است.

اگر تعداد عناصر کم باشد جستجوی خطی به دلیل سادگی از الگوریتم های پیچیده دیگر مناسب تر است. برای لیست های نامرتب اغلب جستجوی خطی اولین انتخاب است. کارائی الگوریتم روی یک لیست مرتب بالا می رود. در این حالت به جای رسیدن به انتهای لیست، جستجو با رسیدن به اولین عنصری که بزرگتر(یا کوچکتر) از آرگومان جستجو است خاتمه پیدا می کند.

جستجوی دودوئی

الگوریتم جستجوی دودوئی (binary search algorithm) روشی برای جستجوی یک مقدار درون یک لیست مرتب است. عنصر وسط لیست انتخاب شده و با آرگومان جستجو مقایسه می شود تا تعیین شود از آن بزرگتر، کوچکتر یا مساوی است. اگر آرگومان از عنصر انتخاب شده بزرگتر باشد جستجو در نیمه پایینی و اگر کوچکتر باشد در نیمه بالائی لیست ادامه پیدا می کند.

کد بازگشتی جستجوی دودوئی به صورت زیر است:

```
int BinarySearch(int A, int value, int low, int high) {
  if (high < low)
    return -1    // not found
  mid = (low + high) / 2
  if (A[mid] > value)
    return BinarySearch(A, value, low, mid-1)
  else if (A[mid] < value)
    return BinarySearch(A, value, mid+1, high)
  else
    return mid    // found
}
```

زمان جستجو $O(\log n)$ است که زمان بهتری نسبت به جستجوی خطی است. اگر آرگومان جستجو برابر با عنصر وسط لیست باشد با یک مقایسه پیدا می شود که بهترین حالت است. در بدترین حالت به $\lfloor \log_2 n \rfloor + 1$ مقایسه نیاز است.

جستجوی دودوئی مثالی از یک الگوریتم تقسیم و غلبه است.

الگوریتم های مرتب‌سازی

از ویکی‌پدیا، دانشنامهٔ آزاد
پرش به: ناوبری, جستجو

الگوریتم مرتب‌سازی، در علوم کامپیوتر و ریاضی، الگوریتمی است که لیستی از داده‌ها را به ترتیبی مشخص می‌چیند.

پر استفاده‌ترین ترتیب‌ها، ترتیب‌های عددی و لغت‌نامه‌ای هستند. مرتب‌سازی کارا در بهینه سازی الگوریتم‌هایی که به لیست‌های مرتب شده نیاز دارند (مثل جستجو و ترکیب) اهمیت زیادی دارد.

از ابتدای علم کامپیوتر مسائل مرتب‌سازی تحقیقات فراوانی را متوجه خود ساختند، شاید به این علت که در عین ساده بودن، حل آن به صورت کارا پیچیده‌است. برای مثال مرتب‌سازی حبابی در سال ۱۹۵۶ به وجود آمد. در حالی که بسیاری این را یک مسئلهٔ حل شده می‌پندارند، الگوریتم کارآمد جدیدی همچنان ابداع می‌شوند (مثلاً مرتب‌سازی کتاب خانه‌ای در سال ۲۰۰۴ مطرح شد).

مبحث مرتب‌سازی در کلاس‌های معرفی علم کامپیوتر بسیار پر کاربرد است، مبحثی که در آن وجود الگوریتم‌های فراوان به آشنایی با ایده‌های کلی و مراحل طراحی الگوریتم‌های مختلف کمک می‌کند؛ مانند تحلیل الگوریتم، داده‌ساختارها، الگوریتم‌های تصادفی، تحلیل بدترین و بهترین حالت و حالت میانگین، هزینهٔ زمان و حافظه، و حد پایین.

فهرست مندرجات

[نهفتن]

- ۱ طبقه‌بندی
- ۲ مرتب‌سازی حبابی
- ۳ مرتب سازی انتخابی
- ٤ مرتب سازی درجی
- ٥ مرتب سازی پایه‌ای) مبنایی(
- ٦ bucket sort
- ۷ مرتب سازی هرمی
- ۸ مرتب سازی شل
- ۹ مرتب سازی سریع
- ۱۰ مرتب سازی ادغامی
- ۱۱ مرتب سازی درجی
- ۱۲ مرتب سازی بوگو
- ۱۳ فهرست الگوریتم‌های مرتب‌سازی
- ١٤ پاورقی
- ١٥ منابع

[ویرایش] طبقه‌بندی

در علم کامپیوتر معمولاً الگوریتم‌های مرتب‌سازی بر اساس این معیارها طبقه‌بندی می‌شوند:

- **پیچیدگی (بدترین و بهترین عملکرد و عملکرد میانگین): با توجه به اندازهٔ لیست (n). در مرتب‌سازی‌های معمولی عملکرد خوب $O(n \log n)$ و عملکرد بد $O(n)$[۲] است. بهترین عملکرد برای مرتب‌سازی $O(n)$ است. الگوریتم‌هایی که فقط از مقایسهٔ کلیدها استفاده می‌کنند در حالت میانگین حداقل $O(n \log n)$ مقایسه نیاز دارند.**
- **حافظه (و سایر منابع کامپیوتر) : بعضی از الگوریتم‌های مرتب‌سازی «در جا[۱]» هستند. یعنی به جز داده‌هایی که باید مرتب شوند، حافظهٔ کمی ($O(1)$) مورد نیاز است؛ در حالی که سایر الگوریتم‌ها به ایجاد مکان‌های کمکی در حافظه برای نگه‌داری اطلاعات موقت نیاز دارند.**
- **پایداری[۲] : الگوریتم‌های مرتب‌سازی پایدار ترتیب را بین داده‌های دارای کلیدهای برابر حفظ می‌کنند. فرض کنید می‌خواهیم چند نفر را بر اساس سن با یک الگوریتم پایدار مرتب کنیم. اگر دو نفر با نام‌های الف و ب هم‌سن باشند و در لیست اولیه الف جلوتر از ب آمده باشد، در لیست مرتب شده هم الف جلوتر از ب است.**
- **مقایسه‌ای بودن یا نبودن. در یک مرتب‌سازی مقایسه‌ای داده‌ها فقط با مقایسه به وسیلهٔ یک عملگر مقایسه مرتب می‌شوند.**
- **روش کلی : درجی، جابجایی، گزینشی، ترکیبی و غیره. جابجایی مانند مرتب‌سازی حبابی و مرتب‌سازی سریع و گزینشی مانند مرتب‌سازی پشته‌ای.**

[ویرایش] مرتب‌سازی حبابی

نوشتار اصلی: مرتب‌سازی حبابی

مرتب‌سازی حبابی (به انگلیسی: *Bubble sort*) یک الگوریتم مرتب‌سازی ساده‌است که لیست را پشت سرهم پیمایش می کند تا هر بار عناصر کنارهم را با هم سنجیده و اگر در جای نادرست بودند جابه‌جایشان کند. دراین الگوریتم این کار باید تا زمانی که هیچ جابه‌جایی در لیست رخ ندهد، ادامه یابد و در آن زمان لیست مرتب شده‌است. این مرتب‌سازی از آن رو حبابی نامیده می‌شود که هر عنصر با عنصر کناری خود سنجیده‌شده و درصورتی که از آن کوچک‌تر باشد جای خود را به آن می‌دهد و این کار همچنان پیش می‌رود تا کوچک‌ترین عنصر به پایین لیست برسد و دیگران نیز به ترتیب در جای خود قرار گیرند (یا به رتبه‌ای بالاتر روند یا به پایین تر لیست رانده شوند.) این عمل همانند پویش حباب به بالای مایع است.

این مرتب‌سازی از آن رو که برای کار با عناصر آن‌ها را با یکدیگر می‌سنجد، یک مرتب‌سازی سنجشی است.

با فرض داشتن n عضو در لیست، در بدترین حالت $n(n-1)/2$ عمل لازم خواهد بود.

فهرست مندرجات

[نهفتن]

- ۱ عملکرد
- ۲ خرگوش‌ها و لاک پشت‌ها
- ۳ مثال قدم به قدم
- ٤ پیاده سازی شبه کد
- ٥ تحلیل
  - ٥.١ بدترین حالت
  - ٥.٢ بهترین حالت
- ٦ دیگر روش‌های پیاده سازی
- ۷ گونه‌های دیگر
- ۸ جستارهای وابسته
- ۹ پیوند به بیرون
- ۱۰ منابع

[ویرایش] عملکرد

بدترین زمان اجرا و پیچیدگی متوسط مرتب سازی حبابی هر دو $O(n^2)$ می‌باشند که در آن n تعداد عناصری است که باید مرتب شوند. الگوریتم‌های مرتب سازی بسیاری وجود دارند که بدترین زمان اجرای آنها از این بهتر است یا پیچیدگی متوسط آنها $O(n \lg n)$ است. حتی بقیه اگوریتم‌های مرتب سازی از $O(n^2)$ مثل [مرتب سازی درجی]، عملکرد بهتری نسبت به مرتب سازی حبابی از خود نشان می‌دهند.

[ویرایش] خرگوش‌ها و لاک پشت‌ها

در مرتب سازی حبابی موقعیت عناصر درون لیست نقش بسزایی در تعیین عملکرد آن دارد. از آنجایی که عناصر بزرگ در ابتدای لیست به سرعت جابجا (swap) می‌شوند، مشکل چندانی در سرعت عملکرد الگوریتم ایجاد نمی‌کنند. اگرچه عناصر کوچک نزدیک به آخر لیست (که باید به سمت ابتدای لیست بیایند) بسیار کند حرکت می‌کنند. این تفاوت در سرعت به حدی است که به عناصر بزرگ، لاک پشت ها، و به عناصر کوچک، خرگوش‌ها می‌گویند.
تلاش بسیاری انجام شده که سرعت حرکت لاک پشت‌ها در مرتب سازی حبابی افزایش یابد. از جمله می‌توان از [Cocktail Sort] نام برد که در این زمینه بسیار خوب عمل می‌کند ولی بدترین زمان اجرای آن هنوز O(n^2) است. مرتب سازی شانه‌ای ([Comb Sort]) عناصر بزرگ با فاصله‌های زیاد را مقایسه می‌کند و لاک پشت‌ها را با سرعت فوق العاده‌ای حرکت می‌دهد سپس با کم تر و کم تر کردن این فاصله‌ها لیست را به سرعت مرتب می‌کند، به طوری که سرعت متوسط آن قابل مقایسه با الگوریتم‌های پر سرعتی مثل [مرتب سازی سریع] ([Quick Sort]) می‌باشد.

[ویرایش] مثال قدم به قدم

اجازه بدهید یک آرایه از عددهای "۵, ۱, ۴, ۲, ۸" اختیار کنیم و آن را به ترتیب صعودی با استفاده از مرتب سازی حبابی مرتب کنیم. در هر مرحله عناصری که در حال مقایسه شدن با یکدیگر هستند پر رنگ تر نشان داده شده اند:

گذر اول:

(۱, ۵, ۴, ۲, ۸) <= (۵, ۱, ۴, ۲, ۸)
در اینجا الگوریتم دو عنصر اول را مقایسه، و جابجا می‌کند.
(۱, ۴, ۵, ۲, ۸) <= (۱, ۵, ۴, ۲, ۸)
(۱, ۴, ۲, ۵, ۸) <= (۱, ۴, ۵, ۲, ۸)
(۱, ۴, ۲, ۵, ۸) <= (۱, ۴, ۲, ۵, ۸)
حالا آرایه مرتب شده است، ولی الگوریتم هنوز نمی‌داند که این کار کامل انجام شده است یا نه، که برای فهمیدن احتیاج به یک گذر کامل بدون هیچ جابجایی (swap) داریم:

گذردوم

(۱, ۲, ۴, ۵, ۸) <= (۱, ۲, ۴, ۵, ۸)
(۱, ۲, ۴, ۵, ۸) <= (۱, ۲, ۴, ۵, ۸)
(۱, ۲, ۴, ۵, ۸) <= (۱, ۲, ۴, ۵, ۸)
(۱, ۲, ۴, ۵, ۸) <= (۱, ۲, ۴, ۵, ۸)
در نهایت آرایه مرتب شده است و الگوریتم می‌تواند پایان پذیرد.

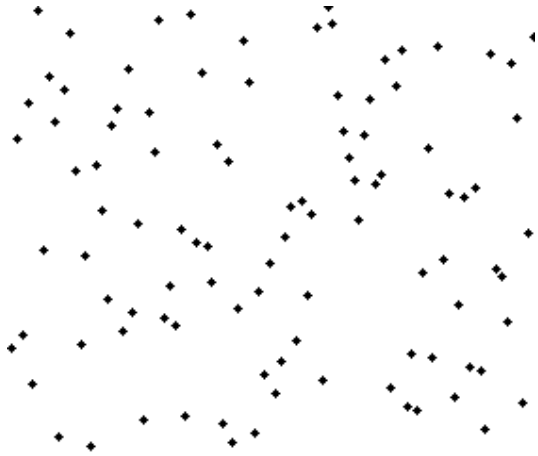
[ویرایش] پیاده سازی شبه کد

بیان سادهٔ شبه کد مرتب‌سازی حبابی :

```
procedure bubbleSort( A : list of sortable items ) defined as:
do
swapped := false
for each i in 0 to length(A) - 2 inclusive do:
if A[ i ] > A[ i + 1 ] then
swap( A[ i ], A[ i + 1 ] )
swapped := true
end if
end for
while swapped
```

end procedure

[ویرایش] تحلیل



نمونه‌ای از مرتب‌سازی حبابی که فهرستی از عددهای تصادفی را مرتب می‌کند.

[ویرایش] بدترین حالت

این الگوریتم در بدترین حالت از مرتبهٔ $\Theta(n^2)$ است. چون در بدترین حالت هر عنصر باید $n-1$ بار فهرست را بپیماید.

[ویرایش] بهترین حالت

بهترین حالت این است که لیست مرتب شده‌باشد که در این حالت الگوریتم از مرتبه $O(n)$ است.

[ویرایش] دیگر روش‌های پیاده سازی

کارایی مرتب سازی حبابی با رعایت شرایط زیر می‌تواند افزایش قابل ملاحظه‌ای داشته باشد. اول این که توجه داشته باشید بعد از هر مقایسه (و احتمالا جابجایی) در هر پیمایش، بزرگ‌ترین عنصری که از آن عبور می کنیم در آخرین موقعیت پیمایش شده قرار خواهد گرفت. از این رو بهد از اولین پیمایش بزرگ‌ترین عنصر آرایه در آخرین خانه آن خواهد بود. این یعنی با داشتن لیستی با اندازه n، پس از اولین پیمایش، n-امین عنصر در مکان نهایی اش خواهد بود. بنا بر استقرا بقیه n-1 عنصر باقیمانده به همین صورت مرتب می‌شوند که بعد از پیمایش دوم، n-1 امین عنصر در جای نهایی خودش خواهد بود، و الی آخر. پس طول هر پیمایش می‌تواندبه اندازه یک مرحله کم تر از پیمایش قبل از خودش باشد و به جای آنکه هر بار تمامی عناصر را تا انتهای آرایه بپیماییم، عناصری که در موقعیت پایانی خود هستند و از به هیچ عنوان جابجا نمی‌شوند چشم پوشی کنیم.

با تبدیل این روش به شبه کد خواهیم داشت:

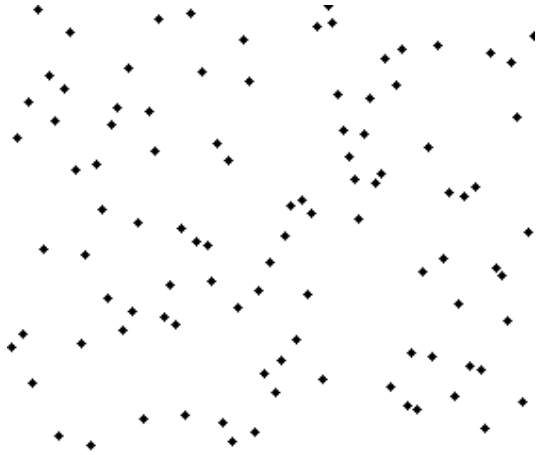
```
procedure bubbleSort( A : list of sortable items ) defined as:
  n := length( A )
  do
    swapped := false
    n := n - 1
    for each i in 0 to n - 1  inclusive do:
      if A[ i ] > A[ i + 1 ] then
        swap( A[ i ], A[ i + 1 ] )
        swapped := true
      end if
```

```
    end for
  while swapped
end procedure
```

زمان اجرای این روش هنوز هم O(n^2) است ولی در بدترین حالت (وقتی آرایه ورودی به صورت معکوس مرتب شده باشد) زمان اجرای آن دو برابر سریع تر از حالت عادی الگوریتم می‌باشد.

[ویرایش] گونه‌های دیگر

مرتب‌سازی زوج-فرد پیاده‌سازی این الگوریتم به شیوهٔ موازی است.



نمونه‌ای از عددهای تصادفی که به وسیلهٔ مرتب‌سازی زوج-فرد مرتب می‌شوند.

[ویرایش] جستارهای وابسته

- **الگوریتم‌های مرتب‌سازی**
- **مرتب‌سازی سنجشی**

[ویرایش] پیوند به بیرون

- **مرتب‌سازی حبابی در ۲۰ زبان برنامه‌نویسی**

[ویرایش] منابع

[ویرایش] مرتب سازی انتخابی

نوشتار اصلی: الگوریتم مرتب‌سازی انتخابی

(به انگلیسی: *Selection Sort*)

معمولاً اطلاعات و داده‌های خامی که در اختیار برنامه نویس قرار داره بصورت نامرتب هستن. مواقعی پیش می‌یاد که لازمه این داده‌ها بر حسب فیلد خاصی مرتب بشن؛ مثل لیست دانش آموزان بر حسب معدل ، لیست کارمندان بر حسب شماره پرسنلی ، لیست دفترچه تلفن بر

حسب نام خانوادگی و ... روشهای متعددی برای مرتب سازی وجود داره که من قصد دارم تا حد امکان شما رو با این روشها آشنا کنم. برای شروع روش مرتب سازی انتخابی (Selection Sort) رو توضیح می‌دم.

روش انتخابی اولین روشیه که به ذهن می‌رسه: بزرگ‌ترین رکورد بین رکوردهای لیست رو پیدا می‌کنیم و به انتهای لیست انتقال می‌دیم. از بقیه رکوردها بزرگ‌ترین رو انتخاب می‌کنیم و انتهای لیست - کنار رکورد قبلی - قرار می‌دیم و ... مثلا:

۰: ۹ ۱ ۶ ۴ ۷ ۳ ۵

۱: ۵ ۱ ۶ ۴ ۷ ۳ ۹

۲: ۵ ۱ ۶ ۴ ۳ ۷ ۹

۳: ۵ ۱ ۳ ۴ ۶ ۷ ۹

۴: ۴ ۱ ۳ ۵ ۶ ۷ ۹

۵: ۳ ۱ ۴ ۵ ۶ ۷ ۹

۶: ۱ ۳ ۴ ۵ ۶ ۷ ۹

پیاده سازی (مرتب سازی انتخابی) در c++

```
void selection_sort (int arr[] , int n)
{
 register int i , j;
 int max , temp;
 (--for (i = n - ۱ ; i > ۰ ; i
 }
  max = ۰;
  for (j = ۱ ; j <= i ; j++)
   if (arr[ max ] < arr[ j])
       max = j;
      ; ] temp = arr[ i
       arr[ i ] = arr[ max];
       arr[ max ] = temp;
 }
}
```

[ویرایش] **مرتب سازی درجی**

نوشتار اصلی: الگوریتم مرتب‌سازی درجی

(به انگلیسی: *Insertion Sort*)

- **در مرتب سازی درجی،ابتدا عنصر دوم با عنصر اول لیست مقایسه می‌شود و در صورت لزوم با عنصر اول جابجا می‌شود به طوری که عناصر اول و دوم تشکیل یک لیست مرتب دوتایی را بدهند. سپس عنصر سوم به ترتیب با دو عنصر قبلی خود یعنی عناصر دوم و اول مقایسه و درجای مناسبی قرار می‌گیرد به طوری که عناصر اول و دوم و سوم تشکیل یک لیست مرتب سوم و دوم و اول مقایسه و درجای مناسب قرار می‌گیرد به طوری که عناصر اول و دوم و سوم و چهارم تشکیل یک لسیت مرتب چهارتایی را بدهند و در حالت کلی عناصر i ام با i-1 عنصر قبلی خود مقایسه می‌گردد تا در مکان مناسب قرار گیرد به طوری که i عنصر تشکیل یک لیست مرتب i تایی را بدهند و این روند تا مرتب شدن کامل لیست ادامه می‌یابد.یا به صورت دقیق تر :**
- **مرحلهٔ ۱:A[۱] خودش به طور بدیهی مرتب است.**
- **مرحلهٔ ۲:A[۲] را یا قبل از یا بعد از A[۱] درج می کنیم طوری که A[۱] و A[۲] مرتب شوند.**
- **مرحلهٔ ۳:A[۳] را در مکان صحیح در A[۱] و A[۲] درج می کنیم به گونه‌ای که A[۱]وA[۲] وA[۳] مرتب شده باشند.**
- **مرحله A[n]:n را در مکان صحیح خود در A[۱]وA[۲] و ... وA[n-1] به گونه‌ای درج می کنیم که کل آرایه مرتب باشد.**
- **زمان اجرای الگوریتم مرتب سازی درجی از(O(n^2 است.**
- **این الگوریتم از الگوریتم های پایدار می‌باشد و در یک آرایهٔ کاملا مرتب بهترین حالت را دارد و برای یک آرایهٔ مرتب شده معکوس بدترین حالت را دارد.**
- **ثابت شده است که برای n های کوچکتر از ۲۰ مرتب سازی درجی سریع ترین روش مرتب سازی است.**

- **پیاده سازی (مرتب سازی درجی) در c++**

```
void Insertion_sort (int A[] , int n)
{
 int i , j ,temp;
  for (i =1 ; i < n ; i++)
  {
    temp = A[i];
    for (j = i ; j >0 && A[j-1]>temp; j--)
      A[j]=A[j-1];
    A[j]=temp;

 }
}
```

[ویرایش] مرتب سازی پایه‌ای (مبنایی)

(به انگلیسی: *radix sort*)

نوشتار اصلی: الگوریتم مرتب‌سازی پایه‌ای

- **مرتب سازی مبنایی الگوریتمی است که لیستی با اندازهٔ ثابت و اعضایی با طول k را در زمـان O(kn) اتجام می‌دهد.ورودی ها را به بخش های کوچکی تقسیم می کنیم(اگر یک کلمه است آن را به حرف هایش می شکنیم و اگر عدد است آن را به ارقامش) سپس ابتدا لیست را بر اساس کم ارزش ترین بیت(حرف یا رقم) مرتب می کنیم، سپس بر اساس دومین بیت ، تا در نهایت بر اساس پرارزش ترین بیت.به این ترتیب پس از k مرحله لیست مرتب می‌شود.**
- **این روش مرتب سازی پایدار است و در تهیهٔ واژه نامه‌ها و مرتب سازی اعداد استفاده می‌شود.**
- **مرتبهٔ اجرایی این الگوریتم در بهترین حالت از(O(nlgn و در بدترین حالت از(O(n^2 است.**
- **پیاده سازی** radix sort

```
{
 int i , j , k ;
for (i = 1 ; i<=5 ; i++)
 {

   for (j = ۰ ; j <n ; j++)
{
    k=ith digit of x[j];
   place x[j] at rear of q[k];
 }
for (j=0;j<10;j++)
place element of q[j] in next sequential position of x;
}
```

[ویرایش] bucket sort

bucket sort به طور متوسط در یک زمان خاصی به طول می انجامد.این الگوریتم با فرض اینکه ورودی ها به طور تصادفی و به صورت یکنواخت در بازهٔ (۱و۰] توزیع شده اند، کار می‌کند.

- **ایدهٔ bucket sort این است که بازهٔ (۱و۰] را به زیربازه‌هایی با سایز یکسان تقسـیم مـی‌کنـد و سـپس ورودیها را در این زیربازه‌ها توزیع می کنیم(در واقع این ورودی ها با توجه به مقدارشان در این زیربازه‌ها قرار می‌گیرند). اگر ورودی ها توزیعی یکنواخت داشته باشند ، انتظار داریم که هر عدد در یک زیربازه قرار گرفته باشد.برای تولید خروجی ، اعداد داخل هر زیربازه را به یک روش مرتب سازی (معمولا مرتب سـازی**

درجی به دلیل کارایی خوب در مرتب سازی تعداد کم ورودی)مرتب می کنیم. سپس داده‌های مرتب شده هر زیربازه در کنار هم قرار می دهیم.

- **شبه کد** bucket sort

1.n<-length [A]

2.for i=1 to n do

3. insert A[i] into list B[nA[i]]

4.for i=0 to n-1 do

5. sort list B with Insertion sort

6.concatenate the list B[0],B[1],...B[n-1] together in order.

[**ویرایش**] **مرتب سازی هرمی**

نوشتار اصلی: الگوریتم مرتب‌سازی هرمی

(به انگلیسی: *Heap Sort*)

همان طور که می دانیم ، هرم تقریبا مرتب است، زیرا هر مسیری از ریشه به برگ ، مرتب است. به این ترتیب ، الگوریتم کارآمدی بـه نـام مرتب سازی هرمی را می‌توان با استفاده از آن به دست آورد.این مرتب سازی همانند سایر مرتب سازی هـا بـر روی یـک آرایـه صـورت می‌گیرد. این روش مرتب سازی همانند مرتب سازی سریع از یک تابع کمکی استفاده می‌کند.

- **پیچیدگی آن همواره از**O(nlgn) **است. و بر خلاف مرتب سازی سریع به صورت بازگشتی نیست.**
- **در این روش درخت** heap **روی آرایه ساخته می‌شود.**
- **این الگوریتم پایدار نمی‌باشد.**

[**ویرایش**] **مرتب سازی شل**

نوشتار اصلی: الگوریتم مرتب‌سازی شل

(به انگلیسی: *Shell Sort*)

نام این الگوریتم از نام مخترع آن گرفته شده‌است. در این الگوریتم از روش درج استفاده می‌شود.

به عنوان مثال رشته f d a c b e را تحت این الگوریتم مرتب می‌کنیم.

F d a c b e : شروع

C b a f d e : مرحله اول

A b c d e f : مرحله دوم

نتیجه : A b c d e f

در مرحله اول : داده‌های با فاصله ۳ از یکدیگر ، مقایسه و مرتب شده ، در مرحله دوم
داده‌های با فاصله ۲ از یکدیگر ، مقایسه و مرتب می‌شوند و در مرحله دوم داده‌ها با
فاصله یک از یکدیگر مقایسه و مرتب می‌شوند .

منظور از فاصله سه این است که عنصر اول با عنصر چهارم(۱+۳) ، عنصر دوم با عنصر پنجم(۵=۲+۳) و عنصر سوم با عنصر ششم(۶=۳+۳) مقایسه شده در جای مناسب خود قرار می‌گیرد .

برای انتخاب فاصله در اولین مرحله ، تعداد عناصر لیست بر ۲ تقسیم می‌شود(/۲n) وفاصله بعدی نیز با تقسیم فاصله فعلی بر ۲ حاصل می‌گردد و الگریتم تا زمانی ادامه پیدا می‌کند که این فاصله به صفر برسد.

برای نمونه اگر تعداد عناصر برابر با ۱۰ باشد ، فاصله در مرحله اول برابر با ۵ ، در مرحله دوم برابر با ۲ ور در مرحله سوم برابر با ۱ و در نهایت برابر با صفر خواهد بود .

زمان مرتب سازی shell از رابطه n پیروی می‌کند که نسبت به ۲n^ بهبود
خوبی پیدا کرده‌است لذا سرعت عمل روش مرتب سازی شل از روشهای انتخابی ، در جی

و حبابی بیشتر است.

پیاده سازی مرتب سازی شل)) در c++ :

```
#include<stdio.h>

#include<conio.h>

< include<string.h#

Void shell(int *,char*,int)
Int main()

{

        Char s[۸۰];

        Int gap[۸۰];

       (); Clrscr

       ;(«: Printf(» Enter a string

     ); Gets(s

     )); Shell(gap,s,strlen(s

     ); Printf(«\n the sorted string is : ٪s»,s

        Getch();

        Return ۰;
```

```
}
**************************//
Void shell(int gap [], char * item, int count)
{
        Register int I, j,step,k,p;
       ; Char x
        Gap[۰] =count /۲;
        While(gap[k] > ۱)
{
 ++; K
Gap[k]=gap[k-۱]/۲;
}//end of while
For (i= ۰;i<=k;i++)
{
Step=gap[i];
For(j=step;j<count; j++)
{
X=item[j];
P=j-step;
      While(p>=۰ &&  x<item[p])
{
Item[p+step]=item[p];
P=p-step;
}
Item[p+step]=x;
   }
}
}
```

[ویرایش] مرتب سازی سریع

نوشتار اصلی: الگوریتم مرتب‌سازی سریع

(به انگلیسی: *Quick Sort*)

مرتب سازی سریع (Quick Sort) از جمله روشهای محبوب و با سرعت بالا برای مرتب کردن داده‌ها محسوب می‌شه. این روش هم مثل روش ادغام از تقسیم و حل (Conqure Divide and) برای مرتب کردن داده‌ها استفاده می‌کنه. به این ترتیب که داده‌ها رو به دو قسمت مجزا تقسیم، و با مرتب کردن اونها کل داده‌ها رو مرتب می‌کنه. برای اینکار یکی از داده‌ها (مثلاً داده اول) به عنوان محور انتخاب می‌شه. داده‌ها بر اساس محور طوری چینش می‌شن که همه داده‌های کوچک‌تر از محور سمت چپ و داده‌های بزرگ‌تر یا مساوی اون سمت راستش قرار می‌گیرن. با مرتب کردن دو قسمت به دست اومده کل داده‌ها مرتب می‌شن. در این حالت مثل روش ادغام نیازی به ادغام کردن داده‌ها نیست. چرا که قسمت سمت راست همگی از قسمت سمت چپ کوچک‌تر هستن و بالعکس. مثلاً اعداد صحیح زیر رو در نظر بگیرید:

۵ ۶ ۱ ۹ -۲ ۴ ۵ ۱۵ ۳ ۱ ۴ ۱۰

عدد ۵ رو به عنوان محور در نظر می‌گیریم. داده‌ها به این صورت بازچینی می‌شن:

۱ -۲ ۴ ۳ ۱ ۴ ۵ ۶ ۹ ۵ ۱۵ ۱۰

همونطور که مشاهده می‌کنید اعداد سمت چپ عدد ۵ زیر خط دار همگی از ۵ کوچیکتر و اعداد سمت راست بزرگ‌تر یا مساوی اون هستن.

پیاده سازی مرتب سازی Quick sort)) در c++

تابع partition با دریافت آرایه و حد بالا و پایین تکه‌ای که باید تقسیم بشه عملیات لازم رو انجام می‌ده، و اندیس محل تفکیک رو (محل عدد ۵ در مثال بالا) به عنوان نتیجه بر می‌گردونه.

```
int partition (int arr[ ] , int low , int high)
{
 int lb = low + ۱ , rb = high , temp , pivot = arr[ low ] , p;
 while (lb <= rb)
 {
  while (arr[ lb ] <= pivot && lb <= rb)
   lb++;
  while (arr[ rb ] > pivot && lb <= rb)
   rb--;
  if (lb < rb)
  {
   temp = arr[ lb];
   arr[ lb ] = arr[ rb];
```

```
  arr[ rb ] = temp;
 }
}
(if (rb == high
 p = high;
else if(rb == low)
 p = low;
else
 p = lb – ١;
arr[ low ] = arr[ p];
arr[ p ] = pivot;
return p;
}
```

اگه این تابع رو برای مثال بالا استفاده کنیم مقدار ۶ (اندیس ۵ زیرخط دار) برگشت داده می‌شه. با تکرار کردن این عملیات برای دو قسمت به دست اومده (در مثال بالا از اندیس صفر تا ۵ و از اندیس ۷ تا ۱۱) داده‌ها به صورت کامل مرتب می‌شن.

بر اساس گفته‌های بالا تابع مرتب سازی به این صورت خواهد بود:

```
void quick_sort (int arr[ ] , int low , int high)
{
if (low < high)
 {
  int p = partition(arr , low , high);
  quick_sort(arr , low , p – ١);
  quick_sort(arr , p + ١ , high);
 }
}
```

همونطور که مشاهده می‌کنید این تابع بصورت بازگشتی نوشته شده. در صورتی که بخواید به صورت غیر بازگشتی بنویسید باید از پشته به صورت زیر استفاده کنید:

```
void quick_sort (int arr[ ] ,int n)
{
```

```
stack st;
st.push(۰);
st.push(n – ۱);
int low , p , high;
while(! st.isempty())
{
 high = st.pop();
 low = st.pop();
 p = partition(arr , low , high);
 if (p > low)
 {
  st.push(low);
  st.push(p – ۱);
 }
 if (p < high)
{
  st.push(p + ۱);
  st.push(high);
 }
}
}
```

[ویرایش] مرتب سازی ادغامی

نوشتار اصلی: الگوریتم مرتب‌سازی ادغامی

(به انگلیسی: *Merge Sort*)

روش مرتب سازی ادغامی از الگوریتم تقسیم و حل (divide-and-conqure) برای مرتب کردن داده‌ها استفاده می‌کنه. در این الگوریتم مساله به چند جزء کوچک‌تر تقسیم می‌شه. هر کدوم از این قسمت‌ها رو به طور مجزا حل کرده ، و با ترکیب اونها به مساله اصلی می‌رسیم. و اما طرح کلی مرتب سازی ادغام:

در این روش داده‌ها به دو قسمت مساوی تقسیم می‌شن. و هر کدوم از این قسمت‌ها - به صورت بازگشتی - مرتب ، و با ادغامشون دادها بصورت کامل مرتب می‌شن.

```
void merge_sort (int arr[ ] , int low , int high)

{

 if (low >= high)
  return;

 int mid = (low + high) / ۲;

 merge_sort (arr , low , mid);

 merge_sort (arr , mid + ۱ , high);

 merge_array (arr , low , mid , high);

}
procedure merge_sort (var arr : array of integer ; l : integer ; h : integer);
var

 m : integer;

begin

 if l >= h then

  exit;

 m := (l + h) div ۲;

 merge_sort (arr , l , m);

 merge_sort (arr , m + ۱ , h);

 merge_array (arr , l , m , h);

end;
```

این توابع اونقدر ساده هستن که نیاز به هیچ توضیحی ندارن. فقط می‌مونه تابع merge_array که دو زیر آرایه رو با هم ادغام می‌کنه.

```
void merge (int arr[ ] , int low , int mid , int high)
{

 register int i , j , k , t;

 j = low;

 for (i = mid + ۱ ; i <= high ; i++)

 {

  while (arr[ j ] <= arr[ i ] && j < i)
```

```
  j++;
 if (j == i)
  break;
 t = arr[ i];
 for (k = i ; k > j ; k--)
  arr[ k ] = arr[ k – ١];
 arr[ j ] = t;
 }
}
procedure merge_array (var arr : array of integer ; l : integer ; m : integer ; h : integer);
var
 i , j , k , t : integer;
begin
 j := l;
 for i := m + ١ to h do
 begin
  while (arr[ j ] <= arr[ i ]) and (j < i) do
   inc (j);
  if j = i then
   break;
  t := arr[ i];
  for k := i downto j + ١ do
   arr[ k ] := arr[ k – ١];
  arr[ j ] := t;
 end;
End;
```

تابع merge_array خود آرایه و اندیسهای بالا ، پایین و جداکننده زیر آرایه‌ای رو که باید ادغام بشه دریافت می‌کنه ، و به صورت درجا (بدون استفاده از آرایه کمکی) دو قمست مرتب شده زیر آرایه رو ادغام می‌کنه.

[ویرایش] **مرتب سازی درجی**

نوشتار اصلی: الگوریتم مرتب‌سازی درجی

(به انگلیسی: *Insertion Sort*)

مرتب سازی درجی یکی از روشهای مرتب سازی رایج و البته نه چندان کارا محسوب می‌شه. این روش در مقایسه با مرتب سازی حبابی و انتخابی سرعت بهتری داره و برای مرتب کردن تعداد کمی از عناصر مناسبه. به همین خاطر مراخل انتهایی روش مرتب سازی سریع با کمک گرفتن از این روش انجام می‌گیره.

الگوریتم مرتب سازی درجی بر اساس مرتب سازیهایی که معمولاً خود ما بصورت دستی انجام می‌دیم طراحی شده. فرض کنید دسته کارتی با شماره‌های ۱ تا ۱۰ بصورت نامرتب و کنار هم روی زمین چیده شدن:

۷۸۶۴۱۰۱۳۹۲۵

کارت دوم رو نسبت به کارت اول در جای مناسب خودش قرار می‌دیم:

۷۸۶۴۱۰۱۳۹۵۲

حالا نوبت به کارت سوم می‌رسه. این کارت رو نسبت به دو کارت قبلی در جای مناسب قرار می‌دیم. چون ۹ در مقایسه با ۲ و ۵ جای درستی داره بدون هیچ جابجایی به کارت چهارم می‌رسیم. جای این کارت رو نسبت به سه کارت قبلی مشخص می‌کنیم:

۷۸۶۴۱۰۱۹۵۳۲

و به همین ترتیب تا آخر ادامه می‌دیم.

اگه n تعداد عناصر رو مشخص کنه ، این روش n - ۱ مرحله رو برای مرتب کردن طی می‌کنه. بعد از اتمام مرحله i ام مطمئنا i + ۱ عنصر اول به صورت مرتب شده هستن (قسمتی که زیرشون خط کشیده شده). این مساله یکی از حسنهای مرتب سازی درجی محسوب می‌شه: در هر مرحله حتما قطعه‌ای از عناصر مرتب شده هستن. مرتب سازی حبابی این ویژگی رو نداره.

پیاده سازی(مرتب سازی درجی) در c++

```
void insertion_sort (int arr[ ] , int n)
{
  register int i , j , t;
 (++ for (i = ۱ ; i < n ; i
  }
  ]; t = arr[ i
  (-- for (j = i ; j > ۰ && arr[ j - ۱ ] >= t ; j
    ; arr[ j ] = arr[ j - ۱]
```

```
    arr[ i ] = t;
  }
}
```

۷ - مرتب سازی Heep Sort))

یک الگوریتم مرتب سازی در حافظه (RAM) می‌باشد. Heap یک درخت دودویی کامل است با ارتفاع Height = ëlog nû هر گره (node) یک کلید بیشتر ندارد که بزرگ‌تر یا برابر کلید گره پدر (parent) می‌باشد. بصورت یک آرایه (Array) ذخیره می‌شود. برای هر گره (i) فرزندان آن در گره‌های (i۲) و (۱i+۲) ذخیره شده‌اند. پدر هر گره (j) در گره (/۲j) می‌باشد.

الگوریتم Insert در Heap Sort چگونه است؟

۱) رکورد جدید در آخر Heap اضافه می‌شود.

۱) کلید آن با کلید گره پدر مقایسه می‌شود و اگر مقدار آن کوچک‌تر بود محل آن با محل گره پدر تعویض می‌شود.

۱) در صورت لزوم عمل (۲) تا ریشه درخت (Root) ادامه می‌یابد.

الگوریتم Remove در Heap Sort چگونه است؟ ۱) کوچک‌ترین کلید که در گره Root می‌باشد خارج می‌شود. ۲) بزرگ‌ترین کلید (آخرین گره) به گره Root منتقل می‌گردد. ۳) کلید آن با کوچک‌ترین کلید فرزند مقایسه می‌شود و اگر بیشتر بود جای آن دو تعویض می‌شود. ۴) در صورت لزوم عمل (۳) تا آخر Heap تکرار می‌گردد.

[ویرایش] مرتب سازی بوگو

نوشتار اصلی: الگوریتم مرتب‌سازی بوگو

(به انگلیسی: *Bugo Sort*)

[ویرایش] فهرست الگوریتم‌های مرتب‌سازی

در این جدول، n تعداد داده‌ها و k تعداد داده‌ها با کلیدهای متفاوت است. ستون‌های بهترین، میانگین و بدترین، پیچیدگی در هر حالت را نشان می‌دهد و حافظه بیانگر مقدار حافظهٔ کمکی (علاوه بر خود داده‌ها) است.

نام	بهترین	میانگین	بدترین	حافظه	پایدار	مقایسه‌ای	روش	توضیحات
مرتب سازی حبابی (Bubble	(O(n	—	۲(O(n	۱(O(	**بله**	**بله**	**جابجایی**	Times are for best variant

sort)								
Cocktail sort	(O(n	—	$O(n)^2$	O(1)	بله	بله	جابجایی	
Comb sort	(O(n log n	—	(O(n log n	O(1)	خیر	بله	جابجایی	
Gnome sort	(O(n	—	$O(n)^2$	O(1)	بله	بله	جابجایی	
Selection sort	$O(n)^2$	$O(n)^2$	$O(n)^2$	O(1)	خیر	بله	گزینشی	
Insertion sort	(O(n	—	$O(n)^2$	O(1)	بله	بله	درجی	
Shell sort	(O(n log n	—	$O(n \log n)^2$	O(1)	خیر	بله	درجی	Times are for best variant
Binary tree sort	(O(n log n	—	(O(n log n	O(1)	بله	بله	درجی	
Library sort	(O(n	(O(n log n	$O(n)^2$	$(1+\varepsilon)O(n)$	بله	بله	درجی	
Merge sort	(O(n log n	—	(O(n log n	(O(n	بله	بله	Merging	
In-place merge sort	(O(n log n	—	(O(n log n	O(1)	بله	بله	Merging	Times are for best variant
Heapsort	(O(n log n	—	(O(n log n	O(1)	خیر	بله	گزینشی	
Smoothsort	(O(n	—	(O(n log n	O(1)	خیر	بله	گزینشی	
Quicksort	(O(n log n	(O(n log n	$O(n)^2$	(O(n	خیر	بله	Partitioning	Naive variants use (O(n

space								
Introsort	(O(n log n	(O(n log n	(O(n log n	**خیر**	**بله**	Hybrid		
Pigeonhole sort	(O(n+k	—	(O(n+k	**بله**	**خیر**	Indexing		
Bucket sort	(O(n	(O(n	$(O(n^2$	**بله**	**خیر**	Indexing		
Counting sort	(O(n+k	—	(O(n+k	**بله**	**خیر**	Indexing		
Radix sort	(O(nk	—	(O(nk	**بله**	**خیر**	Indexing		
Patience sorting	(O(n	—	(O(n log n	(O(n	**خیر**	**بله**	**درجی**	**تمام زیر دنباله‌های صعودی را با** O(n log n)**پیدا می‌کند.**

این جدول الگوریتم‌هایی را توضیح می‌دهد که به علت اجرای بسیار ضعیف و یا نیاز به سخت‌افزار خاص، کاربرد زیادی ندارند.

نام	بهترین	میانگین	بدترین	حافظه	پایدار	مقایسه‌ای	توضیحات
Bogosort	(O(n	O(n × n!)	**بدون حد**	O(1)	**خیر**	**بله**	
Stooge sort	$(O(n^{2.71}$	—	$(O(n^{2.71}$	O(1)	**خیر**	**بله**	
Bead sort	(O(n	—	(O(n	—	N/A	**خیر**	**به سخت افزار مخصوص نیاز دارد.**

Pancake sorting	O(n)	—	O(n)	—	خیر	بله	به سخت افزار مخصوص نیاز دارد.
Sorting networks	O(log n)	—	O(log n)	—	بله	بله	Requires a custom circuit of size O(log n)

ویرایش پاورقی

۱. ^ inplace
۲. ^ stability

ویرایش منابع

- Wikipedia contributors, «Sorting algorithm», Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Sorting_algorithm

نهفتن
ن • ب • و
الگوریتم‌های مرتب‌سازی
تئوری — نظریه پیچیدگی محاسباتی | نماد O بزرگ | ترتیب کلی | پایداری | مرتب‌سازی مقایسه‌ای
مرتب‌سازی تعویضی — مرتب‌سازی حبابی | مرتب‌ساز کوکتل | مرتب‌سازی شانه‌ای | مرتب‌سازی گورزاد | مرتب‌سازی سریع
مرتب‌سازی انتخابی — مرتب‌سازی انتخابی | مرتب‌سازی هرمی | مرتب‌سازی روان | مرتب‌سازStrand
مرتب‌سازی درجی — مرتب‌سازی درجی | مرتب‌سازی شل | مرتب‌سازی درختی | مرتب‌ساز کتابخانه‌ای | مرتب‌سازی شکیبانه
مرتب‌سازی ادغامی — مرتب‌سازی ادغامی
مرتب‌سازی غیرمقایسه‌ای — مرتب‌سازی پایه‌ای | مرتب‌سازی سطلی | مرتب‌سازی شمارشی | مرتب‌سازی لانه‌کبوتری | مرتب‌سازTally
دیگر — مرتب‌سازی توپولوژیکی | شبکه مرتب‌سازی | مرتب‌سازیBitonic
مرتب‌سازی غیرمؤثر — مرتب‌سازیBogo
در ویکی‌انبار منابعی در رابطه با *الگوریتم مرتب‌سازی* موجود است.

برگرفته از

»http://fa.wikipedia.org/wiki/%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«

رده‌های صفحه: الگوریتم‌های مرتب‌سازی
رده پنهان: الگوهای اصلی با پیوند نادرست

- **این صفحه آخرین بار در ۱۴:۵۰، ۱۲ اکتبر ۲۰۱۰ تغییر یافته‌است.**

مراجع:

1. http://www.hpkclasses.ir/Courses/DataStructure/ds1500.html
2. http://fa.wikipedia.org/wiki/%D9%85%D8%B1%D8%AA%D8%A8%E2%80%8C%D8%B3%D8%A7%D8%B2%DB%8C_%D8%AD%D8%A8%D8%A7%D8%A8%DB%8C
3. http://fa.wikipedia.org/wiki/%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