به نام خدا

الگوریتم Timsort

مهدی حق‌وردی

فهرست مطالب

مقدمه و معرفی

مقدمه و معرفی

- در دنیای علوم کامپیوتر، مرتب‌سازی یک عملیات اساسی با کاربردهای بی‌شمار است.
- در میان انبوهی از الگوریتم‌های مرتب‌سازی، یکی از الگوریتم‌ها به دلیل کارایی، تطبیق‌پذیری و طراحی زیبا متمایز شده است: الگوریتم تیم‌سورت[1].
- این الگوریتم که توسط تیم پیترز[2] برای زبان برنامه نویسی پایتون[3] توسعه یافته است، به سنگ بنای پیاده‌سازی مرتب‌سازی در زبان‌ها و محیط‌های مختلف برنامه‌نویسی تبدیل شده است.
- ترکیب منحصر به فرد مرتب‌سازی ادغامی[4] و مرتب‌سازی درجی[5] به همراه بهینه‌سازی‌های مخصوص روی هر الگوریتم و بهینه‌سازی‌های تطبیقی، تیم‌سورت را به یکی از پیچیده‌ترین و کاربری‌ترین الگوریتم‌های مرتب‌سازی موجود تبدیل کرده است.

---

[1] Timsort
[2] Tim Peters
[3] Python programming language
[4] Merge sort
[5] Insertion sort

تاریخچه

تاریخچه

- الگوریتم تیم‌سورت، در سال ۲۰۰۲ توسعه یافت.
- تیم پیترز این الگوریتم را اینگونه توصیف می‌کند:

"A non-recursive adaptive stable natural mergesort / binary insertion sort hybrid algorithm"

- این الگوریتم از Python 2.3 تا حدود بیست سال، الگوریتم استاندارد مرتب‌سازی در پایتون بود و از نسخه‌ی 3.11.1 به دلیل تغییراتی که در سیاست‌های ادغام آن بوجود آمد، الگورتیمی به اسم Powersort بر پایه‌ی تیم‌سورت، جایگزین آن شد.
- الگوریتم تیم‌سورت در Java SE 7، Android، GNU Octave، V8، Swift و Rust پیاده‌سازی شده است.

چراها

- چرا non-recursive؟
چون طبق گفته‌ی تیم پیترز: «به طور خلاصه، روتین اصلی یک بار از سمت چپ تا راست، آرایه را طی، Runها[2] را شناسایی و هوشمندانه آنها را با هم ادغام می‌کند.»

- چرا adaptive؟
چون این الگوریتم با توجه به طول و ترتیب‌های از قبل موجود در آرایه، و همچنین بر اساس اندازه‌ی Runهای پیدا شده، تصمیماتی می‌گیرد تا از الگوریتم بهتری برای آن موقعیت استفاده کند.

- چرا stable؟
چون این الگوریتم، ترتیب عناصر یکسان در آرایه‌ی اولیه را حفظ می‌کند. برای مثال اگر لیستی از این اسامی داشته باشیم: [peach, straw, apple, spork] و آنرا بخواهیم بر اساس حرف اول کلمات مرتب کنیم، چنین چیزی می‌گیریم: [apple, peach, straw, spork] اگر دقت کنید در لیست اولیه، straw قبل از spork آمده بود و در لیست مرتب شده هم همین ترتیب حفظ شد. به این نگهداری ترتیب پایداری الگوریتم مرتب‌سازی می‌گویند.

[2] در ادامه مفهوم Run توضیح داده می‌شود.

چراها (ادامه)

- چرا hybrid؟
چون این الگوریتم از ترکیب دو الگوریتم merge sort و binary insertion sort برای مرتب سازی استفاده می‌کند.

چرا تیم‌سورت؟

چرا تیم‌سورت؟

- پیچیدگی زمانی الگوریتم تیم‌سورت با الگوریتم‌های Merge sort، Quick sort و Heap sort برابری می‌کند و برابر $O(n\lg n)$ است.
- اما این تحلیل کلی یک سری جزئیات راجع به پیچیدگی زمانی الگوریتم را پنهان می‌کند که آن پیچیدگی یک constant factor اثرگذار در میزان پیچیدگی الگوریتم است. $(c_f.n\lg n)$
- برای مثال در الگوریتم Quick sort انتخاب مقدار left، right و pivot تاثیرگذار است و در nهای کوچک سرعت را پایین می‌آورد.
- در الگوریتم Merge sort هم ما فضایی به اندازه‌ی $n + m$ برای ادغام کردن آرایه‌ها آن هم به صورت بازگشتی و تعداد زیاد نیاز دارد. همچنین این الگوریتم یک الگوریتم بازگشتی‌ست و درخت بازگشتی و یک system stack برای اجرا نیاز دارد.
- بخاطر جابجایی‌هایی در الگوریتم Heap sort انجام می‌شود، Locality of Reference در آن نقض شده و پیشبینی‌های پردازنده برای کش کردن داده‌ها را تضعیف می‌کند.

پس اگر بتوانیم این constant factor را کاهش دهیم می‌توانیم سرعت بیشتری از $O(nlgn)$ بگیریم.

مرتب سازی درجی دودویی

- پیچیدگی زمانی insertion sort برابر با $O(n^2)$ است و constant factor آن بسیار بسیار پایین است چون اولا inplace عمل می‌کند (پس نیازی به فضای اضافه ندارد) و ثانیا فقط بین عناصر آرایه پیمایش انجام می‌دهد (پس Locality of Reference هم در آن بسیار خوب است و پردازنده می‌تواند داده‌ها را کش کند).
- در تحلیل‌های انجام شده روی الگوریتم‌ها، این الگوریتم روی تعداد ورودی ۶۴ و پایین‌تر از الگوریتم‌های دیگر مرتب سازی سریع‌تر عمل می‌کند.
- الگوریتم binary insertion sort بجای جستجوی خطی در آرایه (با پیچیدگی $O(n)$) در آن جستجوی دودویی انجام داده و در زمان لوگاریتمی ($O(\lg n)$) مکان صحیح آیتم را پیدا می‌کند (علت استفاده از این الگوریتم در ادامه روشن خواهد شد).

مرتب سازی درجی دودویی (ادامه)

- با تعویض نوع جستجوی این الگوریتم میزان پیچیدگی آن (حالت مورد انتظار و در بدترین حالت) تغییری نکرده و همان $O(n^2)$ باقی می‌ماند؛ اما در CPython مقایسه‌ها (بخاطر ماهیت dynamic typed بودن زبان) نسبت به جابجا کردن آبجکت‌ها بسیار وحشتناک کندتر هستند.
- جابجا کردن آبجکت‌ها صرفا کپی کردن ۸ بایت pointer است اما مقایسه‌ها میتوانند بسیار کند باشند (چون ممکن است چند متد در سطح پایتون را صدا بزنند) و حتی در حالات ساده ممکن است بین ۳ یا ۴ تصمیم گرفته بشود:
  ۱. تایپ عملوند چپ چیست؟
  ۲. تایپ عملوند راست چیست؟
  ۳. آیا باید آنها را به یک تایپ مشخص تبدیل کرد؟
  ۴. چه کدی برای مقایسه این دو موجود هست؟ و...

مرتب سازی درجی دودویی (ادامه)

- پس یک مقایسه ساده باعث تعداد بسیار زیادی C-level pointer dereference، عملیات‌های شرطی و صدا زده شدن توابع می‌شود.
- پس اگر ما تعداد مقایسه‌ها کمتر کنیم میتوانیم سرعت مرتب سازی را بیشتر کنیم (که با استفاده از binary insertion sort ما تعداد مقایسه‌ها را کم می‌کنیم).

ایده

اگر آرایه را به تکه‌های کوچک تقسیم کنیم (برای مثال ۳۲ تا ۶۴ تایی) و سپس آنها را جدا جدا با مرتب سازی درجی مرتب کنیم و سپس همه را ادغام کنیم، می‌توانیم سرعت مرتب سازی را افزایش دهیم.

۱. چون از مرتب سازی درجی که برای تعداد کم سریع است استفاده کردیم ($c_i(32 \text{ to } 64)^2$)،
۲. ادغام دو آرایه مرتب شده در زمان $O(n)$ انجام می‌شود و
۳. با این کار ما توانستیم ۵ سطح از درختی در Merge sort تولید می‌شود را کم کنیم ($c_t.n[\lg n - 5]$)

در نهایت چون مقدار پیچیدگی مرتب سازی درجی کوچک است، پیچیدگی تیم‌سورت چنین می‌شود:

$$T(n) = c_t.n[\lg n - 5]$$

حقیقت

در دنیای واقعی و داده‌های واقعی معمولا آرایه‌ها اصطلاحا partially sorted هستند.

- این به این معناست که تکه‌هایی از آرایه از قبل مرتب هستند؛ برای مثال در این آرایه: $[5, 4, 1, 2, 3]$ قسمت $[1, 2, 3]$ از قبل مرتب است.
- یا حداقل به صورت صعودی یا نزولی پشت سر هم حضور دارند؛ برای مثال در این آرایه: $[6, 4, 1, 2, 3, 5, 7]$ قسمت $[1, 2, 3, 5, 7]$ خودش به صورت صعودی مرتب است.

الگوریتم تیم‌سورت این تکه‌های صعودی و یا اکیدا نزولی را در آرایه پیدا می‌کند و آنها را Run می‌نامد و از مرتب بودن اولیه آرایه برای افزایش سرعت استفاده می‌کند.

نتیجه

- اگر از حقیقت قبلی استفاده کنیم و آرایه را به قسمت‌هایی صعودی و یا اکیدا نزولی تقسیم کنیم می‌توانیم درخت Merge sort حتی بیشتر از قبل هم کوتاه کنیم.
- و پیچیدگی را به $c_t.n[\lg n - x]$ تبدیل کنیم.

توضیح الگوریتم - Runها

Runها

- تابع `count_run()`[1] تعداد عنصر موجود در Run را بر می‌گرداند.
- Runها میتوانند:
  ۱. صعودی باشند: $a_0 \leq a_1 \leq a_2 \leq \ldots$
  ۲. اکیدا نزولی باشند: $a_0 > a_1 > a_2 > \ldots$

- دلیل اینکه یک Run باید نزولی اکید باشد تا یک Run شناخته شود اینست که الگوریتم تیم‌سورت Runهای نزولی را به صورت در جا، برعکس می‌کند[2] و اگر در یک Run نزولی (و نه اکیدا نزولی) دو عنصر یکسان باشند، ماهیت stable بودن الگوریتم نقض می‌شود. برای مثال این آرایه: $[4, 3, 3, 1]$ اگر برعکس شود: $[1, 3, 3, 4]$ ؛ که ترتیب عناصر ۳ عوض شده است. اما اگر اکیدا نزولی باشد دیگر این مشکل وجود نخواهد داشت.

---

[1] https://github.com/python/cpython/blob/3.10/Objects/listobject.c#L1316
[2] https://github.com/python/cpython/blob/3.10/Objects/listobject.c#L1064

Runها (ادامه)

- نکته‌ی دیگر اینست که Runها حداقل دو آیتم دارند مگر وقتی که آخرین عضو آرایه را برای Run جدید برگزینیم.
- اگر عناصر آرایه رندوم باشند، بعید است که ما Runهای طبیعی (یعنی قسمتی از آرایه که از قبل مرتب شده باشد) بلندی را شاهد باشیم. اگر یک Run طبیعی تعداد عناصرش کمتر از minrun باشد (توضیح داده خواهد شد) الگوریتم با استفاده از binray insertion sort تعداد آنرا به حداقل اندازه‌ی Run می‌رساند.
- دلیلی که می‌توانیم از binary insertion sort استفاده کنیم اینست که هر Run خودش مرتب است، پس می‌توان در آن جستجوی دودویی انجام داد.

Runها (ادامه)

- الگوریتم برای پیدا کردن Runها چنین عمل می‌کند: فرض کنید چنین آرایه‌ای داریم: $[8, 12, 9, 17, 15, -1, 22, 11, 10, 7]$ و حداقل اندازه‌ی Runها هم ۳ تعیین شده است،

- از چپ به راست حرکت می‌کنیم و $[8]$ را جدا می‌کنیم و سراغ آیتم بعدی می‌رویم و متوجه می‌شویم که یک Run صعودی داریم: $[8, 12]$ ادامه می‌دهیم و به عدد ۹ می‌رسیم، چون این Run باید صعودی باشد و ۹ از ۱۲ کمتر است با استفاده از binary insertion sort جایگاه ۹ را پیدا می‌کنیم: $[8, 9, 12]$

- عدد بعدی هم بزرگ‌تر از ۱۲ است و آن را هم به این Run اضافه می‌کنیم: $[8, 9, 12, 17]$. عدد بعدی، از ۱۷ کمتر است و چون ما حداقل اندازه‌ی یک Run را داریم آنرا دیگر به این Run اضافه نمیکنیم و به سراغ Run بعدی می‌رویم.

- Run بعدی چنین روندی دارد: $[15]$ سپس $[15, -1]$ و سپس با binary insertion sort: $[22, 15, -1]$ و چون ۱۱ از ۱- بیشتر است و ما حداقل اندازه‌ی یک Run را داریم به سراغ Run بعدی می‌رویم؛ که Run بعدی هم چنین است: $[11, 10, 7]$

Runها (ادامه)

- اگر داده‌های رندوم باشند، اکثر Runها یک اندازه خواهند داشت که دو خوبی دارد:
  ۱. ادغام کردن Runهایی که اندازه‌ی برابر دارند بسیار بهینه است و
  ۲. ما حداقل توانسته‌ایم اندازه‌ی درخت بازگشتی ادغام را به اندازه‌ی $\log(\mathrm{minrun})$ کم کنیم.
- برای داده‌های واقعی هم، ما چون Runهای نسبتا بلندی خواهیم داشت توانسته‌ایم کوتاه‌ترین درخت بازگشتی ادغام را داشته باشیم و در نتیجه تعداد ادغام‌ها را کم کنیم.

محاسبه‌ی minrun

- اگر طول آرایه کمتر از ۶۴ باشد، از binary insertion sort برای مرتب کردن آن استفاده می‌شود.
- اگر طول آرایه توانی از ۲ بود، طبق تست‌های انجام شده تمامی اعداد ۸ و ۱۶و ۳۲ و ۶۴ و ۱۲۸ سرعت یکسانی را به الگوریتم می‌دادند اما مثلا در اندازه‌ی ۲۵۶ تا جابجا کردن عناصر در مرتب سازی هزینه بردار و در اندازه‌ی ۸ تعداد صدا زده شدن توابع هزینه بردار بود. بعد از کمی مطالعه عدد ۳۲ برای minrun انتخاب شد.
- اما بعد از زمان زیادی یک اشکال در انتخاب این عدد پیدا شد، این مثال را ببینید:

```
divmod(2112, 32) -> (66, 0)
```

که اگر با این تعداد Run ما ادغام را انجام بدهیم، در پایان باید یک آرایه‌ی ۲۰۴۸ عضوی و یک آرایه‌ی ۶۴ عضوی را با هم ادغام کنیم که اصلا خوب نیست.

- اما اگر عدد ۳۳ را برای minrun انتخاب کنیم ما ۶۴ تا Run با اندازه‌ی ۳۳ داریم که موقعیت آن بهتر شده است.

محاسبه‌ی minrun (ادامه)

- سیاستی که برای محاسبه‌ی minrun در پیش گرفته شده است اینست که این مقدار از `range(32, 64)` به صورتی انتخاب می‌شود که N/minrun یا دقیقا توانی از دو است، یا اگر این ممکن نبود، اکیدا کمتر از توان ۲ باشد.
- این انتخاب توسط تابع `merge_compute_minrun()`[1] انجام می‌شود.

[1] https://github.com/python/cpython/blob/3.10/Objects/listobject.c#L2012

توضیح الگوریتم - ادغام‌ها

بهتر کردن استفاده از فضا در ادغام

- الگوریتم ادغام در Merge sort الگوریتمی inplace نیست و برای ادغام کردن دو آرایه با طول‌های n و m یک آرایه‌ی جدید به طول $n + m$ نیاز دارد و در کل برای ادغام ما چنین فضایی را اشغال کرده‌ایم:

$$\underbrace{n}_{\text{len}(X)} + \underbrace{m}_{\text{len}(Y)} + \underbrace{n + m}_{\text{new array}}$$

- اما تیم‌سورت برای inplace مرتب کردن و کمتر کردن space overhead الگوریتم مرتب سازی ادغامی از روش بهتری برای ادغام کردن دو Run استفاده می‌کند.

بهتر کردن استفاده از فضا در ادغام (ادامه)

- اگر آرایه‌ای به این صورت داشته باشیم:

$$A = [\underbrace{12, 19, 21, 22}_{X}, \underbrace{3, 5, 17, 22, 107, 109}_{Y}]$$

دو Run عه X و Y در آن می‌توان شناسایی کرد.

- در این روش جدید ادغام ما یک آرایه‌ی موقت به اندازه‌ی Run کوچک‌تر درست می‌کنیم و عناصر Run کوچک‌تر را در آن کپی می‌کنیم. کپی کردن‌های این چنین هم (یعنی یک تکه از یک آرایه) به بهینه‌ترین حالت در سطح پایین اجرا می‌شوند. پس حالا ما چنین چیزی داریم:

$$A = [12, 19, 21, 22, 3, 5, 17, 22, 107, 109]$$
$$T = [12, 19, 21, 22]$$

گام‌ها

- چون آرایه‌ی کوچک‌تر سمت چپ بود، پس از سمت چپ آرایه‌ی موقت و دومین Run شروع به ادغام کردن می‌کنیم. که یعنی:

$$A = [12, 19, 21, 22, 3, 5, 17, 22, 107, 109]$$
$$T = [12, 19, 21, 22]$$

را باهم مقایسه، و برنده (عنصر کوچک‌تر) را در ایندکس صفر آرایه‌ی A می‌گذاریم. که می‌شود:

$$A = [{\color{red}3}, 19, 21, 22, 3, 5, 17, 22, 107, 109]$$
$$T = [12, 19, 21, 22]$$

گام‌ها (ادامه)

- سپس:

$$A = [3, 19, 21, 22, 3, 5, 17, 22, 107, 109]$$
$$T = [12, 19, 21, 22]$$

- که می‌شود:

$$A = [3, 5, 21, 22, 3, 5, 17, 22, 107, 109]$$
$$T = [12, 19, 21, 22]$$

گام‌ها (ادامه)

- سپس:

$$A = [3, 5, 21, 22, {\color{blue}3}, {\color{blue}5}, 17, 22, 107, 109]$$
$$T = [12, 19, 21, 22]$$

- که می‌شود:

$$A = [{\color{red}3}, {\color{red}5}, {\color{red}12}, 22, {\color{blue}3}, {\color{blue}5}, 17, 22, 107, 109]$$
$$T = [12, 19, 21, 22]$$

گام‌ها (ادامه)

- سپس:

$$A = [3, 5, 12, 22, {\color{blue}3}, {\color{blue}5}, 17, 22, 107, 109]$$
$$T = [{\color{blue}12}, 19, 21, 22]$$

- که می‌شود:

$$A = [{\color{red}3}, {\color{red}5}, {\color{red}12}, {\color{red}17}, {\color{blue}3}, {\color{blue}5}, 17, 22, 107, 109]$$
$$T = [{\color{blue}12}, 19, 21, 22]$$

گام‌ها (ادامه)

- سپس:

$$A = [3, 5, 12, 17, 3, 5, 17, 22, 107, 109]$$
$$T = [12, 19, 21, 22]$$

- که می‌شود:

$$A = [3, 5, 12, 17, 19, 5, 22, 22, 107, 109]$$
$$T = [12, 19, 21, 22]$$

گام‌ها (ادامه)

- سپس:

$$A = [3, 5, 12, 17, 19, 5, 17, 22, 107, 109]$$
$$T = [12, 19, 21, 22]$$

- که می‌شود:

$$A = [3, 5, 12, 17, 19, 21, 17, 22, 107, 109]$$
$$T = [12, 19, 21, 22]$$

گام‌ها (ادامه)

- سپس:

$$A = [3, 5, 12, 17, 19, 21, 17, 22, 107, 109]$$
$$T = [12, 19, 21, 22]$$

- که در اینجا عنصر ۲۲ آرایه‌ی موقت انتخاب می‌شود که برای حفظ پایداری الگوریتم مرتب سازی است و ۲۲ای که زودتر آمده را انتخاب کند:

$$A = [3, 5, 12, 17, 19, 21, 22, 22, 107, 109]$$
$$T = [12, 19, 21, 22]$$

گام‌ها (ادامه)

- و حالا که عناصر آرایه‌ی موقت تمام شده، باقی عناصر باقی مانده سر جای خود هستند و دیگر مقایسه و جابجایی لازم نیست و آرایه به صورت مرتب ادغام شد:

$$A = [3, 5, 12, 17, 19, 21, 22, 22, 107, 109]$$

جهت ادغام

- در مثال قبلی Run کوچک‌تر در سمت چپ بود و ما ادغام را از سمت چپ Run بزرگ‌تر و آرایه‌ی موقت انجام دادیم، اما اگر چنین حالتی باشد:

$$A = [\underbrace{3, 5, 17, 22, 107, 109}_{X}, \underbrace{12, 19, 21, 22}_{Y}]$$

- ما چنین چیزی داریم:

$$A = [3, 5, 17, 22, 107, 109, 12, 19, 21, 22]$$
$$T = [12, 19, 21, 22]$$

جهت ادغام (ادامه)

- در این حالت که Run کوچک‌تر در سمت راست بود، ما هم تمام ادغام را از سمت راست انجام می‌دهیم

$$A = [3, 5, 17, 22, 107, 109, 12, 19, 21, 22]$$
$$T = [12, 19, 21, 22]$$

- که می‌شود:

$$A = [3, 5, 17, 22, 107, {\color{blue}109}, 12, 19, 21, {\color{red}109}]$$
$$T = [12, 19, 21, 22]$$

جهت ادغام (ادامه)

- و سپس:

$$A = [3, 5, 17, 22, 107, 109, 12, 19, 21, 109]$$
$$T = [12, 19, 21, 22]$$

- که می‌شود:

$$A = [3, 5, 17, 22, 107, 109, 12, 19, 107, 109]$$
$$T = [12, 19, 21, 22]$$

و الی آخر...

ادغام پیشرفته‌تر با Galloping

تکنیک Galloping

- چنین سناریویی را تصور کنید: ما این آرایه را داریم:

$$A = [\underbrace{1, 2, 3, 6, 10}_{X}, \underbrace{4, 5, 7, 9, 12, 14, 17}_{Y}]$$

- در تکنیک Galloping، ما با استفاده از binary search مکان اولین عضو Y را در X پیدا می‌کنیم که می‌بینیم در جایگاه ۴ باید آن را بگذاریم، **این یعنی تمامی عناصر** $[1, 2, 3]$ **در جایگاه درستی قرار دارند.**

- همین کار را برای آخرین عنصر X نسبت به Y انجام می‌دهیم که مکان آن در جایگاه پنجم دومین Run است، **این یعنی تمامی عناصر** $12, 14, 17$ **در جایگاه درستی قرار دارند.**

- پس ما چنین آرایه‌ای داریم:

$$A = [{\color{red}1, 2, 3}, \underbrace{6, 10}_{X}, \underbrace{5, 7, 9}_{Y}, {\color{red}12, 14, 17}]$$

و صرفا باید الگوریتم ادغامی که بالاتر بحث شد را روی این X و Y کوچک‌تر انجام بدهیم که هم سریع‌تر است و هم فضای کمتری استفاده خواهد کرد.

تکنیک Galloping (ادامه)

- نکته‌ای که در تکنیک Galloping وجود دارد اینست که این روش فقط بعضی وقت‌ها فقط به نفع ماست. به این مثال دقت کنید:

$$A = [\underbrace{1, 3, 5, 7, 9}_{X}, \underbrace{2, 4, 6, 8, 10}_{Y}]$$

- ابتدا مکان ۲ را در X پیدا می‌کنیم که در جایگاه دوم باید قرار بگیرد و سپس مکان ۹ را پیدا می‌کنیم که در جایگاه پنجم باید قرار بگیرد.

- که به چنین آرایه‌ای می‌رسیم:

$$A = [{\color{red}1}, \underbrace{3, 5, 7, 9}_{X}, \underbrace{2, 4, 6, 8}_{Y}, {\color{red}10}]$$

که مشخصا هیچ بهبودی را برای ما نداشت.

تکنیک Galloping (ادامه)

- طبق تست‌هایی که تیم پیترز انجام داده است، تعداد عنصری که باید سر جای خودشان باشند، در ابتدای الگوریتم، باید حداقل ۷ تا باشد که این عدد نام `min_gallop` دارد. این یعنی اگر جایگاه اولین عنصر Y را در X پیدا کنیم و ۷ عنصر در جایگاه خودشان باشند، این تکنیک موثر خواهد بود (همین برای پیدا کردن آخرین عنصر X در Y هم صدق می‌کند).
- در ادغام کردن اگر حالت اول (مکان اولین عنصر Y در X) **یا** حالت دوم (مکان آخرین عنصر X در Y) بزرگ‌تر مساوی `min_gallop` بود، از این تکنیک استفاده می‌شود.
- بعد از هر بار موفق بودن galloping عدد `min_gallop` به عنوان یک تشویق برای الگوریتم، یکی کم می‌شود تا در ادامه هم بتوان از galloping استفاده کرد.

تکنیک Galloping (ادامه)

- حالت عکس هم چنین است که مکان پیدا شده کمتر از `min_gallop` باشد که یعنی این تکنیک موثر نبوده و از حالت مقایسه‌ی ادغامی ساده، یعنی عنصر به عنصر استفاده می‌شود.
- روش galloping در داده‌های واقعی که partially sorted هستند بسیار کارآمد است و
  ۱. تعداد مقایسه‌ها را کم و
  ۲. از کپی کردن بهینه سطح پایین استفاده می‌کند.
- اما در داده‌های رندوم این روش موثر نیست و از مقایسه‌ی عنصر به عنصر استفاده می‌شود که
  ۱. کد ساده‌ای دارد و
  ۲. Locality of Reference خوبی هم دارد.