دانشکده ریاضی و علوم کامپیوتر

برنامه‌سازی پیشرفته و کارگاه

الگوریتم‌های سورت

استاد درس

دکتر مهدی قطعی

استاد دوم

بهنام یوسفی مهر

نگارش

پریا اصحابی و هومن حمیدی پور

بهار ۱۴۰۳

فهرست

مقدمه ای بر الگوریتم ها و پیچیدگی زمانی

الگوریتم‌ها همان دستورالعمل‌های مرحله‌به‌مرحله‌ای هستن که تو زندگی روزمره و حل مسائل همیشه ازشون استفاده می‌کنیم، از کارهای ساده‌ای مثل مسواک زدن تا چیزای پیچیده‌تر مثل برنامه‌نویسی. وقتی می‌خوایم به دانشگاه بریم یا یه مسئله برنامه‌نویسی رو حل کنیم، در واقع داریم از یه الگوریتم استفاده می‌کنیم، فقط ممکنه خودمون آگاه نباشیم! نکته‌ی جالب اینه که برای هر کاری همیشه فقط یه راه‌حل وجود نداره و می‌شه با روش‌های مختلفی به جواب رسید، بعضی روش‌ها سریع‌ترن، بعضی‌ها کم‌هزینه‌تر و بعضی‌ها ساده‌تر. همین تنوع راه‌حل‌هاست که بهمون انگیزه می‌ده تا همیشه دنبال روش‌های بهینه‌تر بگردیم و کارهامون رو بهتر انجام بدیم.

یکی از مهم‌ترین معیارهای ما برای مقایسه الگوریتم‌ها سرعتشونه. اما چطور می‌تونیم این مقایسه رو انجام بدیم؟ زمان اجرای یه برنامه فقط به الگوریتمی که باهاش نوشته شده وابسته نیست و به خیلی چیزای دیگه، مثل سخت‌افزاری که برنامه روش اجرا می‌شه، زبان برنامه‌نویسی و غیره هم بستگی داره. پس زمان اجرایی یه الگوریتم ممکنه بسته به نحوه نوشتن و اجرا شدنش متفاوت باشه. اما چیزی که بین همه این‌ها ثابت می‌مونه، تعداد عملیات‌هاییه که از اول تا آخر الگوریتم انجام می‌شه تا به جواب برسه...

شمردن تعداد عملیات ها

بنظر می‌آد که بخاطر محدودیت هایی که راجع بهشون حرف زدیم نمی‌تونیم واقعا زمان بگیریم و ببینیم الگوریتم هامون توی چه زمانی اجرا می‌شن، اما اینکه بشمریم که کامپیوتر ما چند تا عملیات انجام می‌ده می‌تونه راه خوبی برای مقایسه متوسط زمان اجرای الگوریتم هامون باشه. فرض کنید می‌خوایم مجموع اعداد 1 تا n رو بدست بیاریم. کد زیر اعداد یک تا n رو باهم جمع می‌کنه و حاصل رو خروجی میده. بیاین باهم تعداد عملیات های معناداری که انجام می‌ده رو بشمریم:

```
public class Main {
    public static void main(String[] args) {

        int n = readInput();
        int i = 1;
        int sum = 0;
        while (i <= n) {
            sum += i;
            i++;
        }
        System.out.println(sum);
    }
}
```

بیاین ببینیم این کد چند تا عملیات انجام می‌ده. اول متغیرها رو مقداردهی می‌کنیم (3 تا عملیات)، بعد توی حلقه n بار شرط رو چک می‌کنیم و هر بار چند تا عملیات (مثل جمع) انجام می‌دیم. اگه تعداد دقیق عملیات های معنادار رو بشمریم، به $3n + 5$ تا عملیات می‌سیم. ولی حالا یه سوال: این عدد واقعاً به ما چی می‌گه؟

اینجاست که یه کم به مشکل می‌خوریم. فرض کنین همین الگوریتم رو یه جور دیگه بنویسیم:

```
public class Main {
    public static void main(String[] args) {

        int n = readInput();
        int i = 1;
        int sum = 0;
        while (i <= n) {
            int temp = sum;
            temp += i;
            sum = temp;
            i++;
        }
        System.out.println(sum);
    }
}
```

می‌بینیم که تعداد عملیات‌ها عوض می‌شه، و این بار $5n + 5$ تا عملیات داریم ولی مگه ما داریم الگوریتم متفاوتی اجرا می‌کنیم؟ نه! هنوز داریم اعداد رو دونه‌دونه جمع می‌کنیم. پس انگار با شمردن عملیات‌ها، بیشتر داریم پیاده‌سازی‌ها رو مقایسه می‌کنیم، نه خود الگوریتم رو.

حالا بیاید زمان اجرای متوسط همین دوتا برنامه رو باهم مقایسه کنیم فرض کنید به الگوریتمتون مقدار 10^4 رو ورودی بدید، و این دوتا برنامه به ترتیب 30005 و 50005 تا عملیات انجام می‌دن. بسته به سخت افزارتون ممکنه این مقدار متفاوت باشه، ولی متوسط تعداد عملیات هایی که کامپیوتر های ما می‌تونه توی 1 ثانیه انجام بده حدود 10^8 تا عملیاته. یعنی جفت این برنامه ها، جوابشونو توی کمتر از یک هزارم ثانیه خروجی می‌دن پس اختلاف زمان خروجی دادنشون $2 * 10^{-4}$ هستش و این مقدار واقعا ناچیزه. در ادامه این قضیه رو خیلی بیشتر می‌بینین، ولی اختلاف تعداد 5n +5 و 3n + 5 تا عملیات خیلی ناچیز تر از اختلاف n و n^2 تا عملیات هستش. درواقع اگه به برنامه‌ای که n^2 تا عملیات انجام می‌ده همین ورودی 10^4 رو بدیم، محاسبه کردنش حدود 1 ثانیه طول می‌کشه که اختلافش با دوتا برنامه قبلی خیلی بیشتره.

پس دیدیم با اینکه این روش خوبی های خودش رو داشت، مقداری که از شمردن تعداد عملیات ها بدست می‌آریم علاوه بر اینکه خیلی بهمون اطلاعات نمی‌داد، درواقع سرعت پیاده‌سازی‌ها رو نشون می‌داد و تشخیص اینکه آیا این الگوریتم کندِ یا مشکل از پیاده‌سازیمون بوده همچنان قابل تشخیص نیست. در ادامه قراره با یه مفهومی آشنا بشیم که مشکلات شمردن دقیق عملیات هارو حل می‌کنه.

پیچیدگی زمانی

پیچیدگی زمانی مفهومیه که برای مقایسه سرعت الگوریتم‌هامون ازش استفاده می‌کنیم و تعریف دقیق ریاضی داره که تو درس ساختمان داده کامل بررسی می‌شه. اما اگه بخوایم به زبان ساده باهاتون آشنا کنیم، می‌تونیم این مثال رو بزنیم:

اگه ازتون بپرسم وزن یه هندونه چقدره، چی جواب می‌دین؟ احتمالاً اگه وزن دقیقش رو ندونین، می‌گین "چند کیلو". حالا اگه وزن یه ماشین رو بپرسم چی؟ احتمالاً می‌گین "چند تن". چرا نگفتیم "چند کیلو"؟ اگه "چند" که در واقع ضریب "کیلو" هست رو حدود ۱۰۰۰ در نظر بگیریم، می‌شه وزن ماشین رو به کیلو گفت. ولی شما به‌طور طبیعی برداشتتون از "چند" یه ضریب منطقی و معقوله که نسبت به اختلاف واحدهامون (که یکیشون ۱۰۰۰ برابر اون یکیه) خیلی کوچیکتره. اینجا برامون مهم نیست وزن

هندونه دقیقاً چقدره؛ فقط کافیه بدونیم یه ضریب منطقی از "کیلو" هست و با "چند تن" خیلی فاصله داره.

مسئله مجموع اعداد ۱ تا n رو یادتونه؟ الگوریتممون برای حلش، جمع کردن اعداد با یه حلقه بود که بدنه‌ش n بار تکرار می‌شد. تعداد عملیات‌هایی که برای دو تا پیاده‌سازیمون به دست آوردیم، $3n + 5$ و $5n + 5$ بود. از دید پیچیدگی زمانی، برامون فرقی نداره که تو اون n بار حلقه، ۳ تا کار انجام می‌دیم یا ۵ تا؛ هر دوشون نسبت به‌n های بزرگ خیلی کوچیکن و در نهایت یه ضریب منطقی از n تا عملیاتن. حتی $3n$ و $3n + 5$ و $3n + 10000$ هم از نظر پیچیدگی زمانی فرقی باهم ندارن. وقتی n خیلی بزرگ باشه، یه ضریب از n تا عملیات انجام می‌دیم که خیلی بزرگتر از اون عدد ثابتیه که باهاش جمع می‌کنیم و این عدد ثابت عملاً تأثیری نداره. پس می‌گیم پیچیدگی زمانی این الگوریتم از $O(n)$ (خوانده می‌شه اُردر n) هست.

حالا اگه همین مسئله رو با یه الگوریتم دیگه حل کنیم چی؟ کد زیر رو ببینین:

```
public class SumFormula{
    public static void main(String[] args) {

        int n = readInput();
        System.out.println(n * (n + 1) / 2);
    }
}
```

همون‌طور که می‌بینین، دیگه اعداد رو دونه‌دونه جمع نمی‌کنیم و با یه فرمول ریاضی، با چند تا عملیات مجموع ۱ تا n رو به دست می‌آریم. پیچیدگی زمانی این الگوریتم از $O(1)$ هست، چون تعداد عملیات‌هاش هیچ ربطی به اندازه ورودی نداره و همیشه چند تا عملیات ثابت (۵ تا) انجام می‌ده. حتی اگه n خیلی بزرگ باشه، باز با همون چند تا عملیات جواب رو می‌گیرم، در حالی که الگوریتم قبلی برای هر عدد بین ۱ تا n چند تا عملیات انجام می‌داد و تعداد عملیات‌هاش یه ضریب از n می‌شد.

حالا اگه بخوایم همین مسئله رو به شکل دیگه حل کنیم، مثلاً به‌جای اضافه کردن مستقیم i به مجموع، یه حلقه داشته باشیم که i بار اجرا بشه و هر بار ۱ به مجموع اضافه کنیم:

```
public class NestedLoopSum{
    public static void main(String[] args) {

        int n = readInput();
        int i = 1;
        int sum = 0;
        while (i <= n) {
            int j = 1;
            while(j <= i){
                sum++;
```

```
                j++;
            }
            i++;
        }
        System.out.println(sum);
    }
}
```

این بار یه حلقه داریم که n بار اجرا می‌شه و هر بار یه حلقه دیگه رو کامل اجرا می‌کنیم. اینجا چند تا مفهوم مهم رو یاد می‌گیریم:

اول اینکه پیچیدگی زمانی یه کران بالا برای تعداد عملیات‌هاست. اگه دقت کنین، حلقه داخلی هر بار فقط i بار تکرار می‌شه. این i از ۱ شروع می‌شه و تو آخرین تکرار حلقه خارجی، i برابر n هست. ولی چون کران بالا حساب می‌کنیم، می‌تونیم فرض کنیم حلقه داخلی هر بار n بار تکرار می‌شه.

دوم اینکه وقتی یه حلقه تو حلقه دیگه داریم، عملیات‌های حلقه داخلی (که حدود n تاست) به‌خاطر حلقه خارجی n بار تکرار می‌شن. پس در کل حدود n^2 تا عملیات انجام می‌دیم و پیچیدگی زمانی این الگوریتم از $O(n^2)$ هست. دقت کنین که تعداد عملیات‌ها دقیقاً n^2 نیست و در واقع $an^2 + bn + c$ تا عملیاته، ولی همون‌طور که قبلاً گفتیم، وقتی an^2 عملیات داریم (که از O(n^2) هست)، اضافه کردن bn (از O(n)) یا c (از O(1)) تأثیری روی $O(n^2)$ نمی‌ذاره.

تا اینجا یه مقدار به صورت مفهومی با پیچیدگی زمانی آشنا شدیم و دلیل همه قاعده های پایین رو کم و بیش فهمیدیم ، ولی اگه بخوایم یک مقدار این قاعده‌ها رو دقیق تر بیان کنیم:

قواعد پیچیدگی زمانی

- پیچیدگی زمانی عملیات های ساده مثل مقداردهی یک متغیر، عملیات های اصلی، دسترسی به یکی از خونه های یک آرایه و... از O(1) هست.
- ضرایب تاثیری در پیچیدگی زمانی ندارن و می‌تونن حذف بشن: O(3n) = O(n)
- توی جمع پیچیدگی زمانی دو تا الگوریتم، پیچیدگی زمانی کل برابر با پیچیدگی زمانی بیشتر هستش: O(5n + 3) = O(5n)
- اگه دستورات دوتا الگوریتم مختلف پشت سر هم اجرا بشن، پیچیدگی زمانی کل می‌شه مجموع پیچیدگی زمانی الگوریتم ها: مثلا اگه دوتا حلقه مستقل از هم داشته باشیم که پیچیدگی زمانی هرکدوم از O(n) باشه، پیچیدگی زمانی الگوریتممون می‌شه مجموع پیچیدگی زمانی دو حلقه.

- اگه با هربار وارد شدن به بدنه یک حلقه، یک الگوریتم اجرا بشه، پیچیدگی زمانی حلقه برابر با تعداد تکرار های حلقه، ضربدر پیچیدگی زمانی الگوریتم داخلیه: مثلا اگه دوتا حلقه تو در تو داشته باشیم، و هرکدوم از حلقه ها n بار تکرار بشن، با فرض اینکه پیچیدگی زمانی حلقه داخلی از O(n) هستش، پیچیدگی زمانی حلقه بیرونی از $O(n^2)$ خواهد بود.

اندازه ورودی

آخرین مفهومی که تو این بخش بهش اشاره می‌کنیم، اندازه ورودی هست. تو مسئله‌ای که درباره‌ش حرف می‌زدیم، ورودی فقط یک عدد n بود و ما بررسی کردیم اگه ورودی‌مون n باشه، بسته به الگوریتم، پیچیدگی زمانی چطور به n وابسته‌ست. شما قراره پیچیدگی‌های زمانی مثل O(n) رو زیاد ببینین، ولی همیشه ورودی‌ها به این شکل نیستن که فقط یک عدد n بهتون بدن.

ورودی ممکنه یک آرایه باشه و مسئله این باشه که آرایه رو سورت کنین یا چیزای دیگه. از اونجایی که معمولاً هر چی حجم ورودی بیشتر باشه، الگوریتم عملیات‌های بیشتری انجام می‌ده، پیچیدگی زمانی الگوریتم‌ها یک تابع از اندازه داده‌های ورودی‌ست. ما باید بسته به مسئله تشخیص بدیم اندازه ورودی‌مون چیه و با افزایش اون، تعداد عملیات‌ها چطور زیاد می‌شه، و بعد پیچیدگی زمانی رو بر اساسش حساب کنیم.

مثلاً تو مسئله سورت کردن یک آرایه، اندازه ورودی طول آرایه‌ست و اگه پیچیدگی زمانی یک الگوریتم $O(n^2)$باشه، اون n تعداد عناصر آرایه‌مونه.

ارتباط ساختمان داده ها با الگوریتم

اگه یادتون باشه هفته پیش وقتی درباره ساختمان داده‌ها باهاتون حرف می‌زدیم، می‌گفتیم هر کدومشون برای یه هدفی طراحی شدن و مزایا و معایب خودشون رو دارن. همون‌طورکه دیدیم چند تا الگوریتم مختلف می‌تونن سرعت و پیچیدگی‌های زمانی متفاوتی برای حل یه مسئله داشته باشن، تو ساختمان داده‌های مختلف هم نحوه ذخیره داده‌ها باعث می‌شه پیچیدگی زمانی دسترسی یا تغییر داده‌ها فرق کنه. برای حل مسائل، بسته به نیاز مسئله، انتخاب ساختمان داده مناسب می‌تونه پیچیدگی زمانی الگوریتم رو بهترکنه.

اون موقع هنوز با پیچیدگی زمانی آشنا نبودین، ولی حالا می‌تونین این رو تحلیل کنین که تو آرایه، دسترسی به خونه‌های آرایه با O(1) انجام می‌شه، ولی اگه بخوایم یه عنصر رو حذف کنیم و عناصر بعدی رو یکی به چپ شیفت بدیم، O(n) تا عملیات لازم داریم.

برعکس، تو یه لینکد لیست، اگه آبجکت مورد نظرمون رو از قبل داشته باشیم، می‌تونیم با O(1)حذفش کنیم. ولی اگه بخوایم به یه عنصر، مثلاً عنصر وسط لینکد لیست، دسترسی پیدا کنیم، باید با O(n) تا عملیات دونه‌دونه از هر عنصر به بعدی بریم تا به عنصر مورد نظر برسیم.

برای درک بیشتر، می‌تونین پیچیدگی زمانی متدهای دیگه ساختمان داده‌هایی که هفته پیش یاد گرفتین رو تحلیل کنین.

چرا مرتب‌سازی انقدر مهمه؟

بیاین یه لحظه تصور کنین یه لیست بلندبالا از نمره‌های امتحان دانشجوها دارین، یا یه عالمه داده تو یه دیتابیس که باید به ترتیب خاصی نشون داده بشن. حالا اگه بخواین این داده‌ها رو مرتب کنین، چی کار می‌کنین؟ می‌تونین دستی بشینین و یکی‌یکی مقایسه کنین، ولی اگه تعداد داده‌ها مثلاً ۱۰۰۰ تا یا ۱ میلیون تا باشه چی؟ اینجا دیگه دستاتون از خستگی قفل می‌کنه! مرتب‌سازی یکی از اون مسائل کلاسیک علوم کامپیوتره که انگار همه‌جا پیداش می‌شه: از جست‌وجو تو گوگل بگیر تا مرتب کردن لیست آهنگ‌های پلی‌لیستتون.

حالا سوال اینه: چطور می‌تونیم این کار رو سریع و بهینه انجام بدیم؟ مثل خیلی از مسائل زندگی، برای مرتب‌سازی هم یه راه‌حل واحد وجود نداره. یه عالمه الگوریتم مختلف داریم که هرکدوم یه جور به ماجرا نگاه می‌کنن. بعضی‌هاشون مثل دونده‌های استقامت، آهسته و پیوسته پیش می‌رن، بعضی‌ها مثل ماشین‌های مسابقه‌ای سریعن ولی بنزین زیادی می‌خوان (یعنی حافظه!). فلسفه پشت این الگوریتم‌ها یه جورایی مثل انتخاب بهترین مسیر برای رسیدن به دانشگاهه: یه مسیر ممکنه کوتاه باشه ولی پر ترافیک، یکی دیگه طولانی‌تره ولی خلوت. تو این بخش قراره چند تا از این الگوریتم‌های معروف رو ببینیم، بفهمیم چطور کار می‌کنن، و کجاها به کارمون میان. آماده‌این که یه کم ذهنتون رو قلقلک بدیم؟

چند سورت معروف و کاربردی

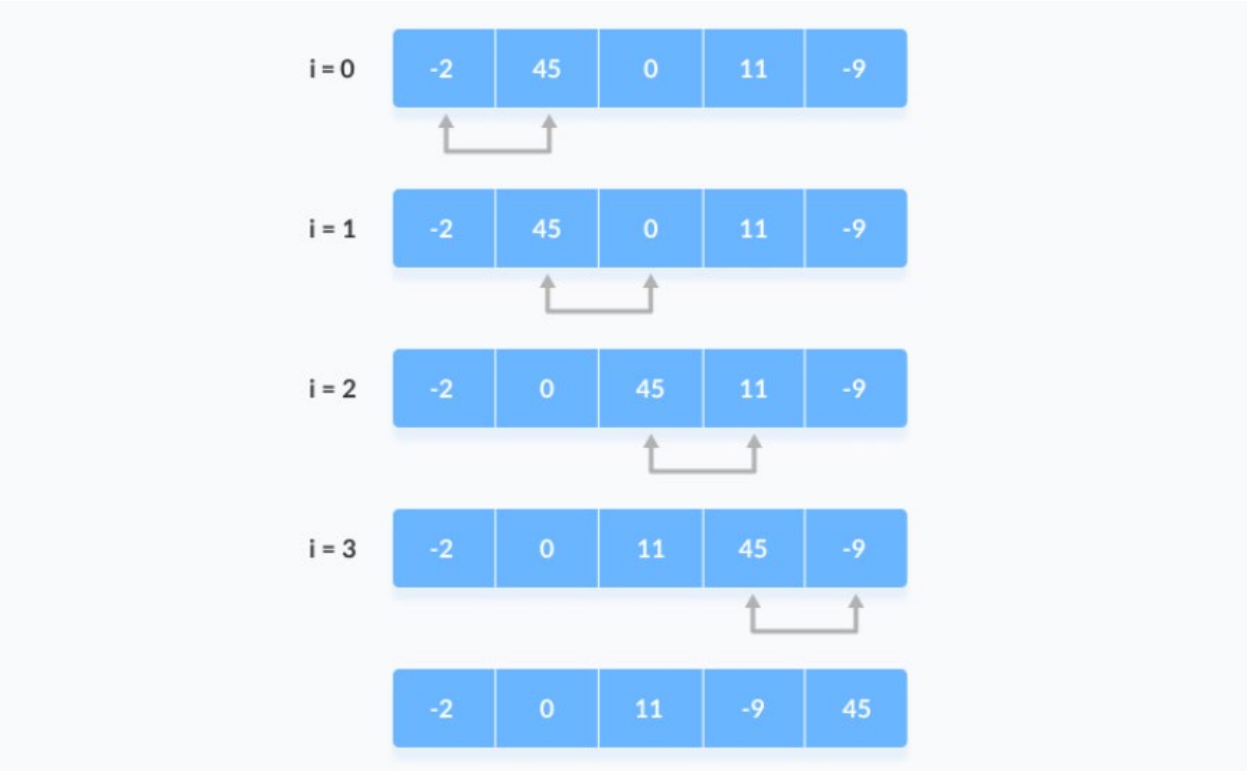
خب حالا که حسابی برای مرتب‌سازی کنجکاو شدین، بریم چند تا الگوریتم سورت معروف و پیاده‌سازیشون رو ببینیم که بفهمیم چطور کار می‌کنن و هر کدوم کجاها کاربرد دارن.

Bubble sort(مرتب‌سازی حبابی):

تصور کنین یه لیوان آب پر از حباب‌های ریز دارین. حباب‌های بزرگ‌تر (یا سنگین‌تر) سریع‌تر به سطح آب می‌رسن و بالا می‌آن. تو این الگوریتم هم دقیقاً همین اتفاق می‌افته. هر بار دو تا عدد مجاور رو مقایسه می‌کنیم و اگه عدد چپی از راستی بزرگ‌تر بود، جابه‌جاشون می‌کنیم. با این کار، بعد از n بار تکرار، بزرگ‌ترین عدد می‌ره آخر آرایه.

به مثال زیر دقت کنید:

مرحله اول:

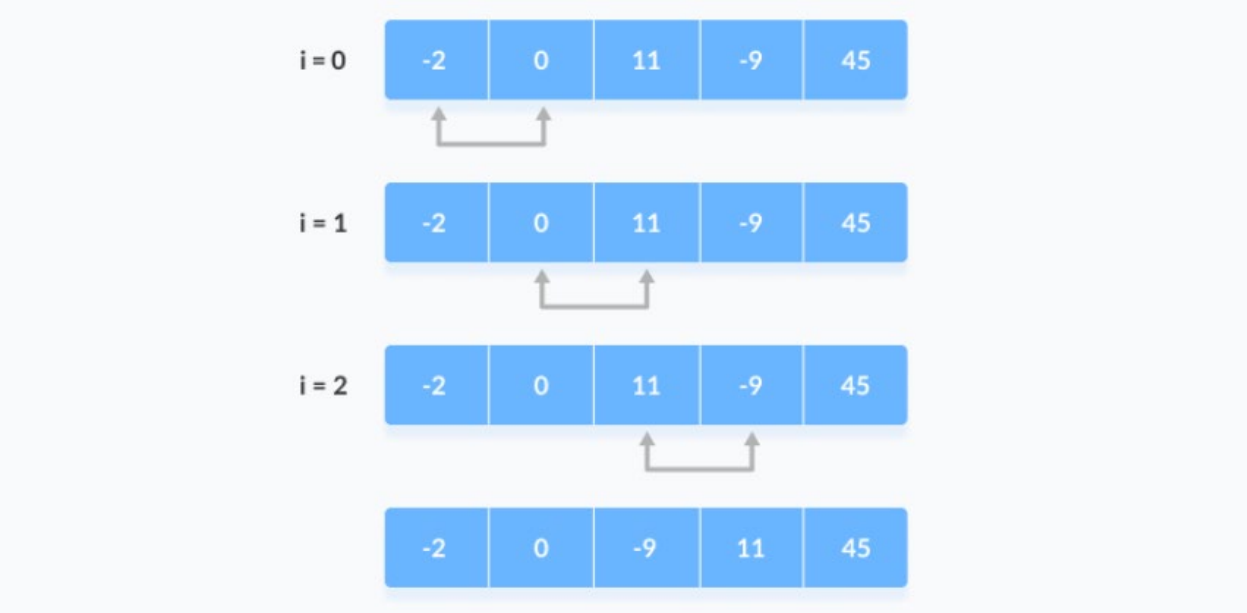


ابتدا عنصر اول و دوم را مقایسه می‌کنیم 2- از 45 کوچکتره پس جابجایی نداریم. سپس عنصر دوم و سوم را مقایسه می‌کنیم چون 0 از 45 کوچکتره، پس اون‌ها رو جابجا می‌کنیم.

به این ترتیب تا عنصر n-1ام و nام را مقایسه می‌کنیم و در آخر کار بزرگترین عدد در ایندکس آخر قرار می‌گیره.

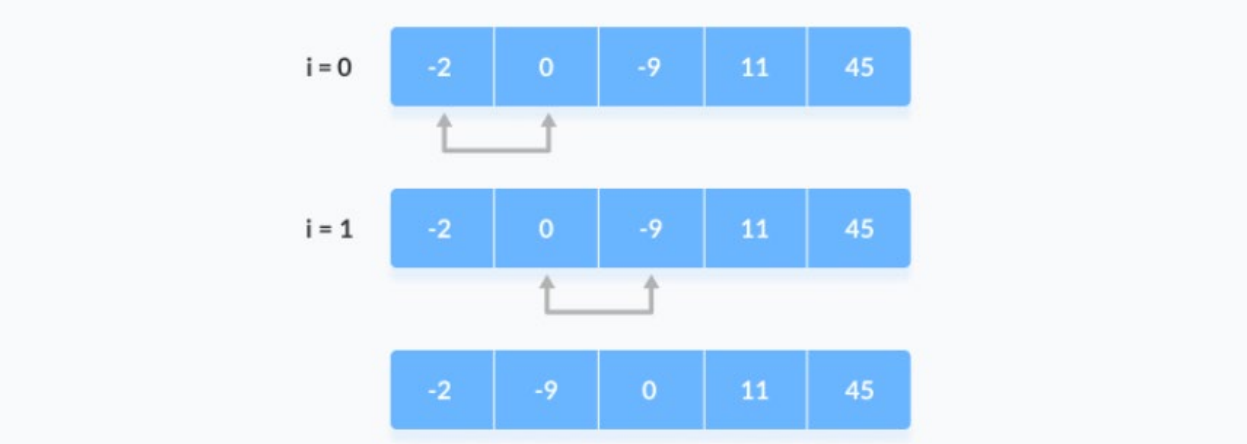
حالا اگر این‌کار را دوباره با عناصر اول تا n-1ام انجام بدیم، دومین بزرگترین عدد در جایگاه یکی مونده به آخر قرار می‌گیره.

مرحله دوم:

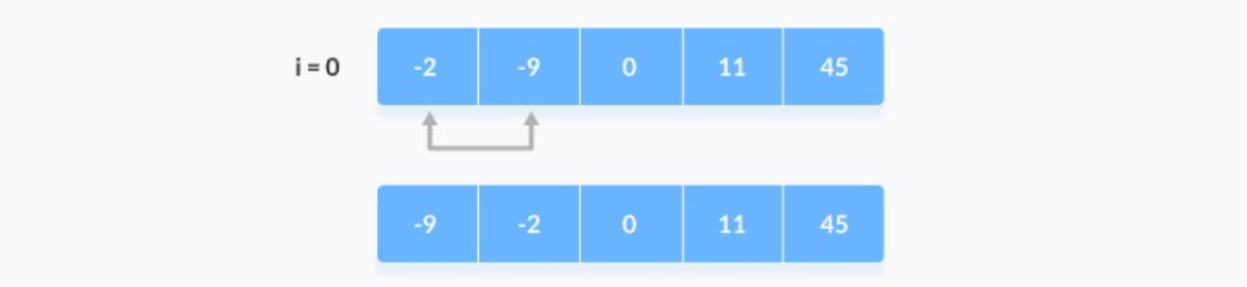


به این ترتیب در هر مرحله i، iامین بزرگترین عدد می‌ره سر جای درستش و بعد از n بار انجام این مراحل آرایه کامل مرتب می‌شه.

مرحله سوم:



مرحله چهارم:



پس از اجرای این مراحل آرایه به صورت سورت شده درمی‌آد.

حالا بریم پیاده‌سازی این الگوریتم رو به زبان جاوا ببینیم:

```
static void bubbleSort(int arr[]) {
    int size = arr.length;
    for (int i=0;i < size - 1; i++){
        for (int j=0;j<size-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1]=temp;
            }
        }
    }
}
```

می‌بینیم که اینجا حدودا $(n) * \frac{n-1}{2}$ عملیات انجام شده، پس این الگوریتم از $O(n^2)$ است.

مزایا:

۱. برای فهمیدن و پیاده‌سازی ساده‌ست.

۲. نیاز به حافظه اضافه نداره (جزو الگوریتم‌های In-place هست).

معایب:

۱. کُندی وحشتناک برای داده‌های بزرگ (برای nهای بیشتر از 1000 توصیه نمی‌شه).

Selection Sort (مرتب‌سازی انتخابی):

این روش مثل این می‌مونه که ده این کار رو آن‌قدر تکرار می‌کنی تا همه عددها تو جای درستشون قرار بگیرن.

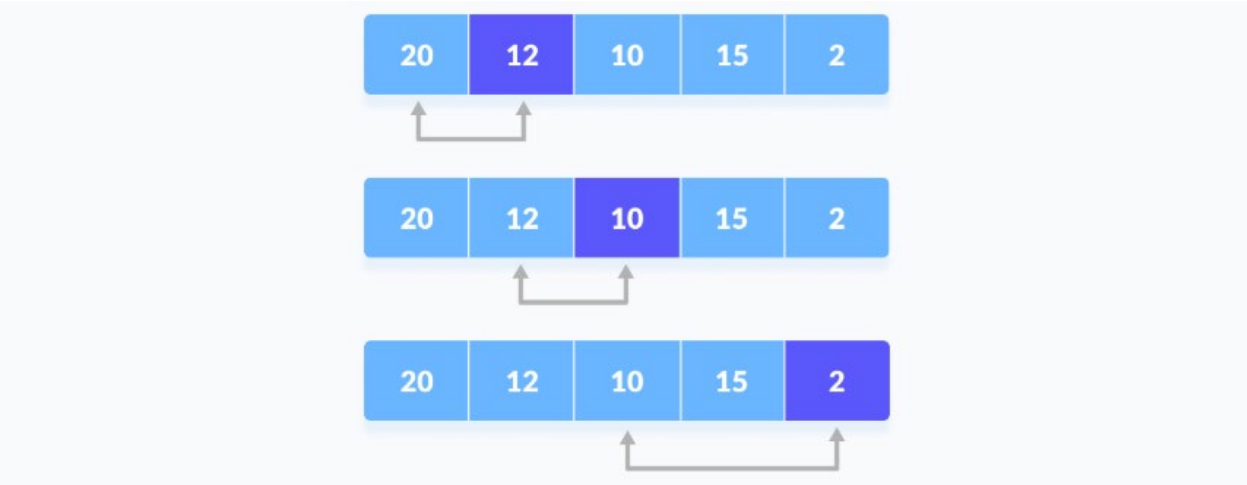
الگوریتم این‌جوری کار می‌کنه: یه بار کل آرایه رو پیمایش می‌کنیم، کوچک‌ترین عدد رو پیدا می‌کنیم و تو خونه اول می‌ذاریم. بعد n−1 خونه باقی‌مونده (همه خونه‌ها به‌جز اولی) رو نگاه می‌کنیم، کوچک‌ترینشون رو پیدا می‌کنیم و تو خونه دوم می‌ذاریم. همین کار رو برای n−2 خونه بعدی تکرار می‌کنیم. می‌تونیم ببینیم که بعد از n−1 بار، هر عدد سر جاشه و آرایه مرتب شده.

بریم یه مثال ببینیم تا الگوریتمو بهتر یاد بگیریم:

مرحله اول:

اول عدد اول رو به عنوان مینیمم در نظر می‌گیریم

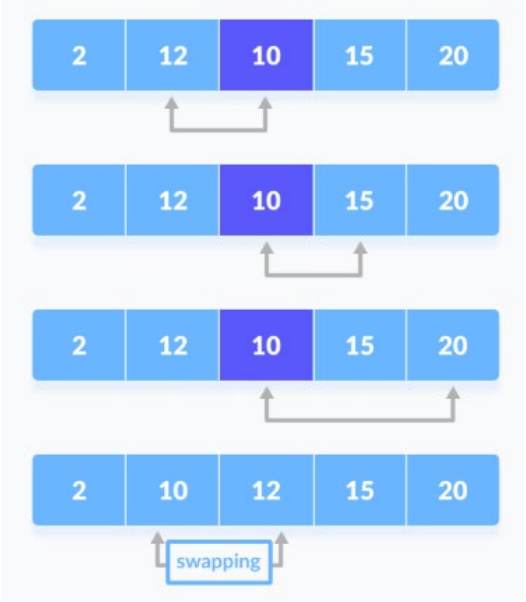
حالا عنصر مینیمم رو با عدد دوم مقایسه می‌کنیم، اگه عدد دوم کوچیکتر بود اون رو به عنوان مینیمم قرار می‌دیم. همین مقایسه رو با عناصر سوم چهار و تا nام انجام می‌دیم تا مینیمم کل آرایه رو پیدا کنیم.



حالا که مینیمم رو پیدا کردیم اون رو با عنصر اول جابجا می‌کنیم.

همین کار را برای عناصر دوم تا nام تکرار می‌کنیم

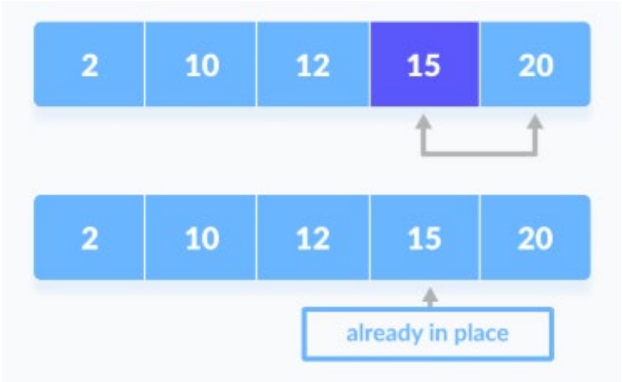
مرحله دوم:



مینیمم اون‌هارو پیدا می‌کنیم و تو خونه دوم می‌ذاریم.

مرحله سوم:

مرحله چهارم:



پس از چهار مرحله آرایه به صورت سورت شده در اومد.

حالا بریم پیاده‌سازی این الگوریتم رو به زبان جاوا ببینیم:

```
public static void selectionSort(int[] arr) {
    int size = arr.length;
    for (int i=0;i<size-1;i++) {
        int minIndex = i;
        for (int j=i+1;j < size;j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}
```

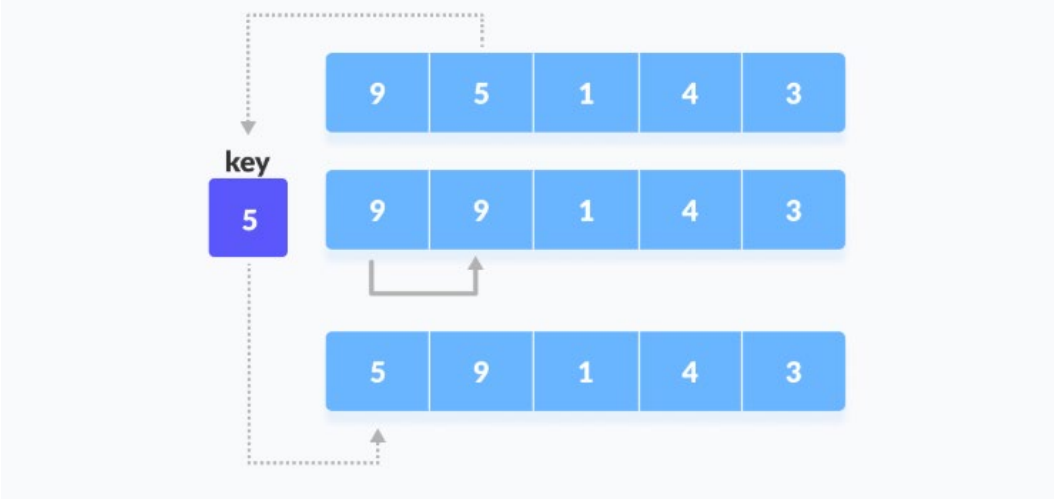
پیچیدگی زمانی و مزایا و معایب این سورت دقیقاً مثل Bubble Sort ه:$O(n^2)$ ، ساده ولی کند برای داده‌های بزرگ.

Insertion sort (مرتب‌سازی درجی):

فرض کنین یه دسته کارت بازی به‌هم‌ریخته دارین و می‌خواین به ترتیب عددی مرتبشون کنین. این الگوریتم دقیقاً مثل اینه که کارت‌ها رو یکی‌یکی برمی‌دارین و تو جای درستشون تو دستتون می‌ذارین. هر کارت جدید رو با کارت‌های قبلی مقایسه می‌کنین و اون‌قدر به چپ می‌لغزونین تا به جای مناسب برسه.

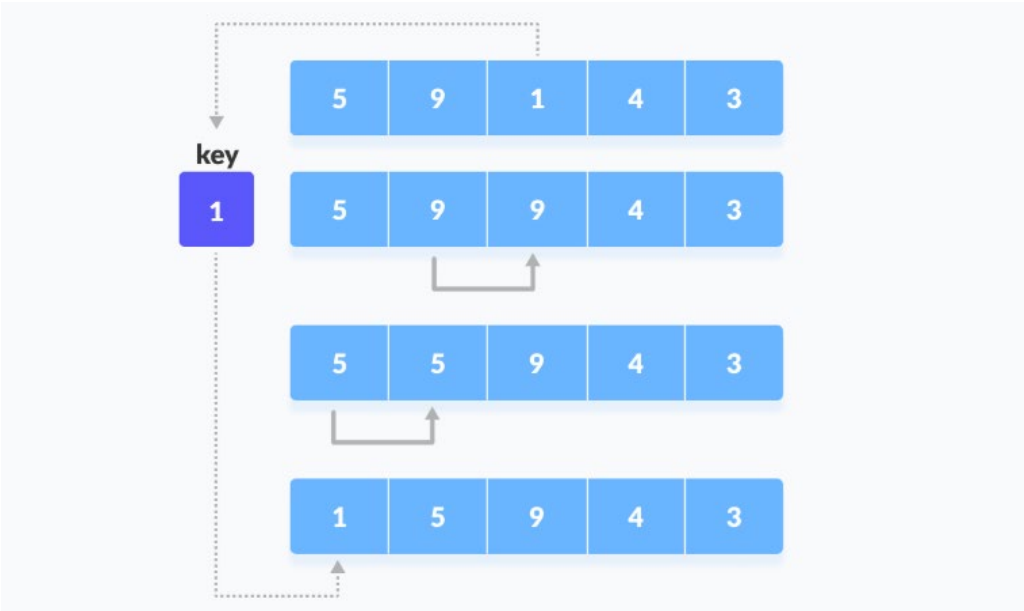
مرحله اول:

فرض می‌کنیم اولین عنصر آرایه از قبل مرتبه. عنصر دوم رو برمی‌داریم و به عنوان "کلید" یا یک عنصر جدا نگه می‌داریم. این کلید رو با اولین عنصر مقایسه می‌کنیم، اگه اولین عنصر بزرگتر بود، کلید رو جلوش می‌ذاریم. حالا دو تا عنصر اول مرتبن.



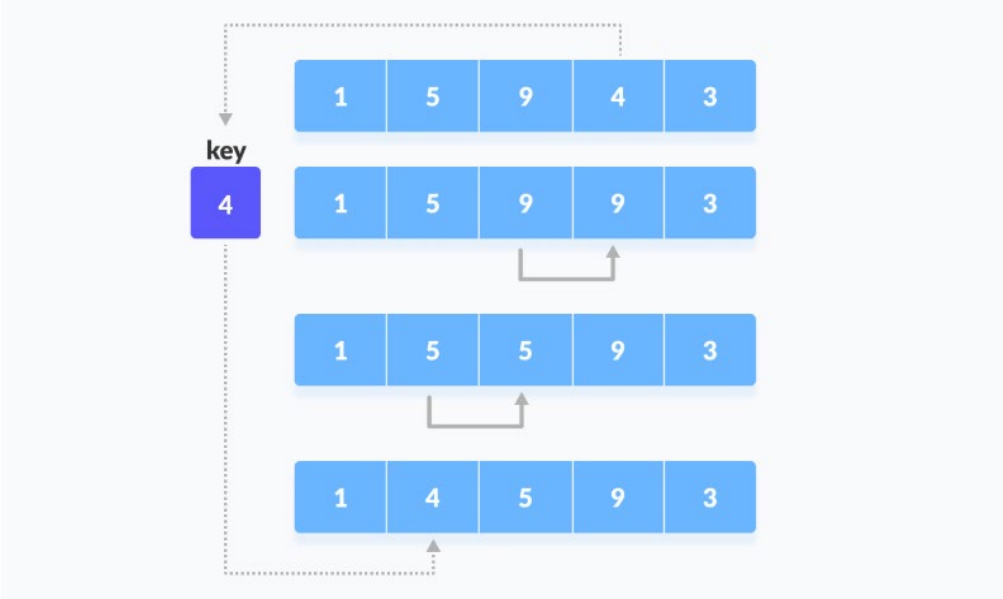
مرحله دوم:

حالا عنصر سوم رو برمی‌داریم و با عنصرهای سمت چپ‌اش (همونایی که قبلاً مرتب شدن) مقایسه می‌کنیم. اونقدر به چپ می‌ریم تا به یه عنصری برسیم که از کلید کوچیکتر باشه. کلید رو دقیقا جلوی اون عنصر می‌ذاریم (اگه هیچ‌کدوم از عناصر چپ از کلید کوچیکتر نبودن، کلید رو اول آرایه می‌ذاریم).

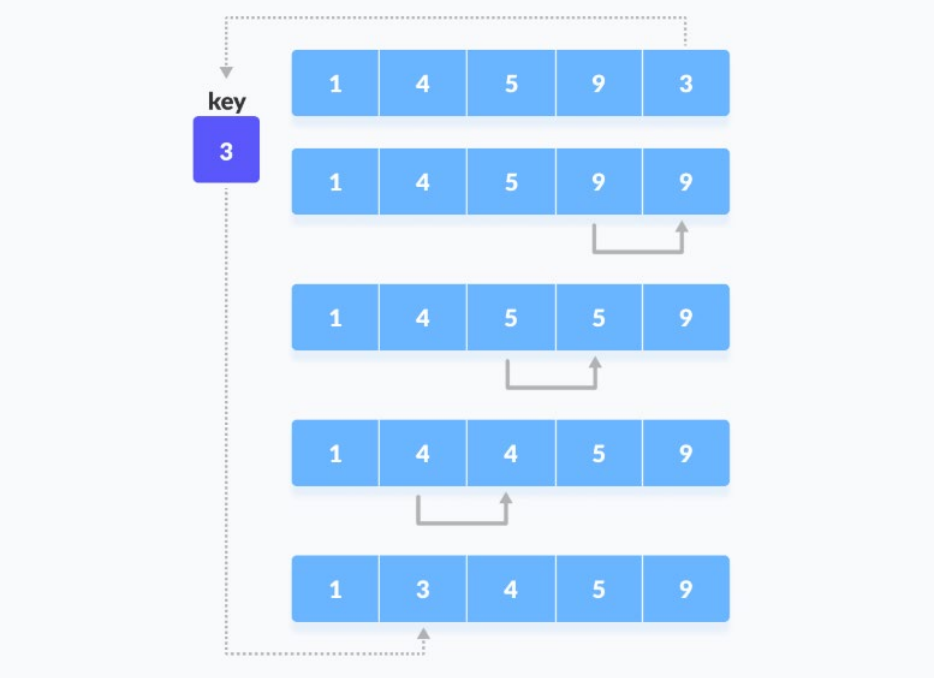


هر مرحله دوباره همین کار را تکرار می‌کنیم. هرسری اولین عنصر از قسمت سورت نشده رو انتخاب می‌کنیم و کلید را برابر آن قرار می‌دهیم. سپس آنقدر در قسمت سورت شده به چپ می‌رویم تا جایگاه درست آن را پیدا کنیم.

مرحله سوم:



مرحله چهارم:



حال می‌بینیم که آرایه به صورت سورت شده در آمده است.

حالا بریم پیاده‌سازی این الگوریتم رو به زبان جاوا ببینیم:

```
void insertionSort(int array[]) {
    int size = array.length;
    for (int i=1;i<size;i++) {
        int key= array[i];
        int j= i-1;
        while (j>=0 && key < array[j]) {
            array[j+1] = array[j];
            j--;
        }
        array[j+1]= key;
    }
}
```

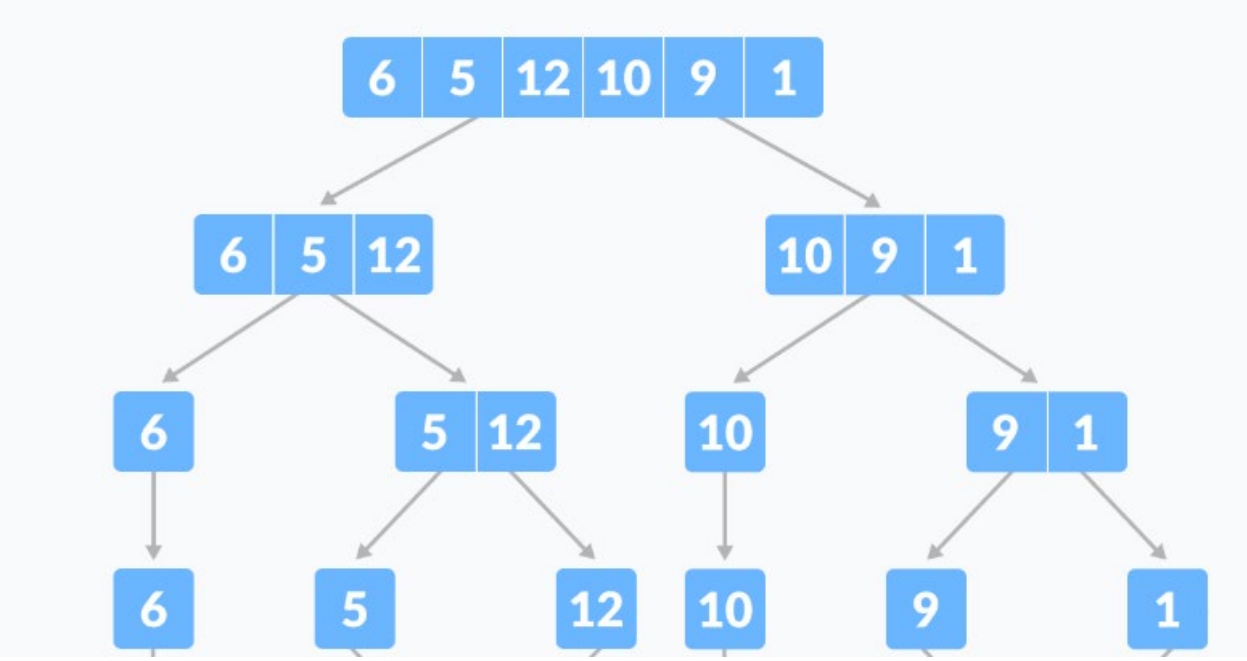
می‌توان دید که اردر زمانی این الگوریتم هم از $O(n^2)$ است. همچنین این الگوریتم هم جزو الگوریتم‌های In-place است پس مزایا و معایب آن نیز مانند دو الگوریتم قبلی است.

Merge sort(مرتب‌سازی ادغامی):

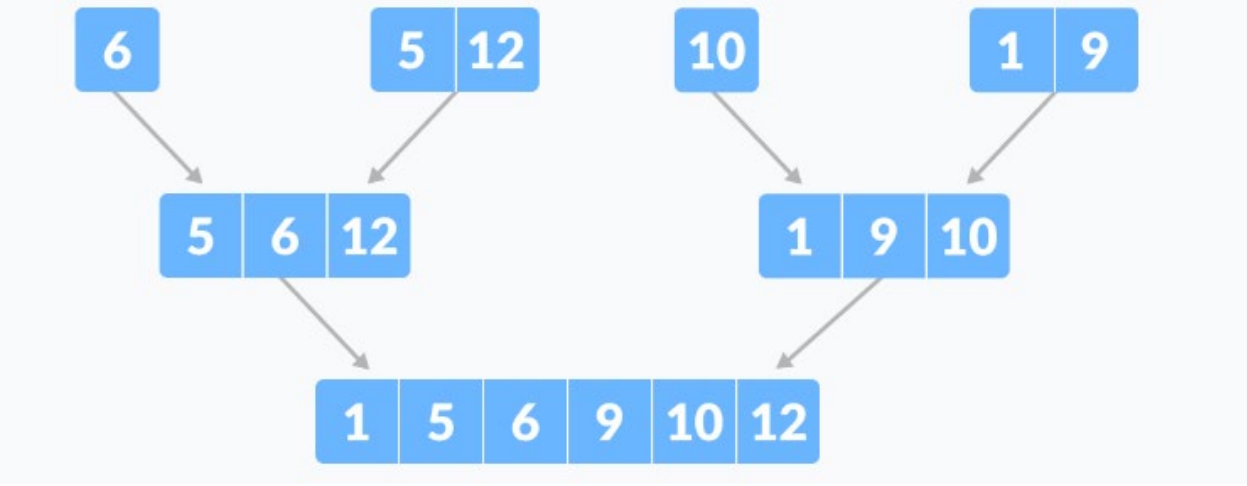
فرض کنین یه آرایه دارین که اونو به دو قسمت تقسیم می‌کنین و هر قسمت رو جداگونه مرتب می‌کنین. اگه این دو قسمت رو ساده پشت سر هم بذارین، آرایه‌تون لزوماً مرتب نیست (یه مثال نقض براش پیدا کنین!). ولی اگه بتونین این دو قسمت رو یه جور خوب ادغام کنین که آرایه مرتب بشه چی؟ حالا اگه برای مرتب کردن هر قسمت، دوباره نصفشون کنین و این کار رو ادامه بدین چی؟ این الگوریتم قراره همین کار رو یادمون بده.

مرج سورت جزو الگوریتم‌های تقسیم و حله. یعنی چی؟ یعنی مسئله رو به زیرمسئله‌های کوچک‌تر تقسیم می‌کنیم، هر کدوم رو حل می‌کنیم، و آخر سر با ادغام اینا جواب مسئله اصلی رو می‌سازیم.

اول آرایه رو نصف می‌کنیم، هر نصف رو دوباره نصف می‌کنیم، و این کار رو ادامه می‌دیم تا به عناصر تکی برسیم.



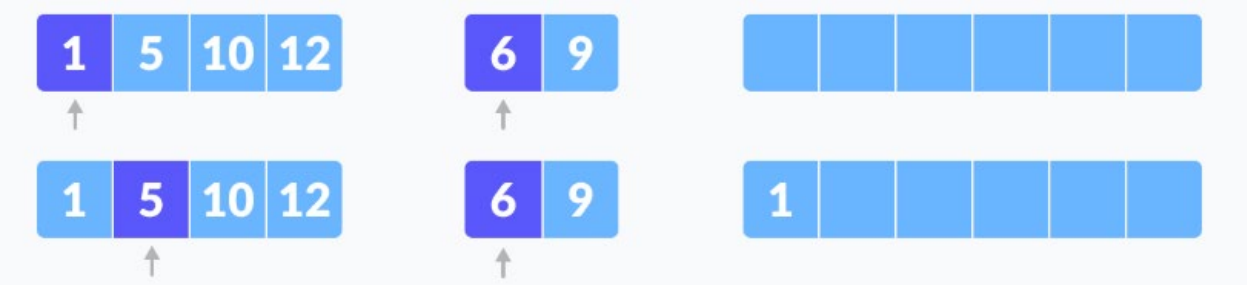
بعد انگار قراره همه‌چیز رو برعکس کنیم: هر بازه‌ای که نصف کردیم رو دوباره با هم ادغام می‌کنیم، ولی جوری که چپی همیشه از راستی کوچک‌تر باشه و زیربازه‌های مرتب‌شده موقع ادغام همچنان مرتب بمونن.



حالا چطور ادغام کنیم؟

فرض کنین دو تا زیربازه مرتب‌شده دارین، هرکدوم یه اشاره‌گر رو عنصر اولشون دارن، و یه آرایه خالی که قراره نتیجه ادغام توش بره.

- عناصر اشاره‌گرها رو مقایسه می‌کنین، کوچک‌تره رو تو آرایه خالی می‌ذارین و اشاره‌گرش رو یکی جلو می‌برین.



حالا دوباره دوتا عنصری که اشاره‌گر روشون هست رو مقایسه می‌کنیم (در این مثال 6 و 5) اونی که کوچیک تره رو می‌ذاریم تو خونه دوم آرایه و پوینتر زیربازه مربوط به آن را یکی جلو می‌بریم. به ترتیب انقدر این‌کار را انجام می‌دهیم تا تمام عناصر یکی از زیربازه‌ها تمام شود سپس تمام عناصر اونیکی زیربازه رو هم به همان ترتیب خودش در انتهای آرایه قرار می‌دیم.

حالا که فهمیدیم الگوریتم چطور کار می‌کنه، بیایین کدش رو بزنیم. چون مدام نصف می‌کنیم و ادغام می‌کنیم، بهترین روش کد زدن به‌صورت بازگشتیه.

به این صورت که یک متد mergeSort داریم که هرسری خودش را برای دو زیربازه‌اش صدا می‌زنه، این دو زیربازه رو سورت می‌کند سپس با یک متد مرج، این دو زیربازه را مرج می‌کند.

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right)/2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

حالا متد merge چجوری کار می‌کنه؟ دقیقا همونطوری که قبل‌تر توضیح دادیم. ابتدا دوتا آرایه جدید موقت می‌سازیم تا زیربازه‌ها رو توش نگه داریم بعد برای هرکدوم یک اشاره‌گر قرار می‌دیم.

```
void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int[] leftArr = new int[n1];
    int[] rightArr = new int[n2];

    for (int i = 0; i < n1; i++) {
        leftArr[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        rightArr[j] = arr[mid + 1 + j];
    }

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
```

پیچیدگی زمانی این الگوریتم O(n log n) هست، چون log n بار بازه‌ها رو نصف می‌کنیم و هر مرحله برای ادغام حداکثر n عملیات داره.

مزایا:

۱ .چون پیچیدگیش از O(n^2) بهتره، برای داده‌های بزرگ خیلی سریع‌تر از الگوریتم‌های قبلیه.

معایب:

۱ . حافظه بیشتری مصرف می‌کنه (برای زیربازه‌ها آرایه موقت می‌سازیم).

۲ . الگوریتم آن برای کدنویسی سخت‌تر از الگوریتم‌های قبلیه.

دسته بندی های مختلف الگوریتم های مرتب سازی

حالا که با چند تا سورت معروف مثل حبابی، انتخابی، درجی، و ادغامی آشنا شدین و مزایا و معایبشون رو دیدین، بیایین یه نگاهی به ویژگی‌های مختلف این الگوریتم‌ها بندازیم. این ویژگی‌ها تو بعضی سناریوها خیلی مهم می‌شن و کمک می‌کنن بفهمیم هر الگوریتم کجا به‌درد می‌خوره. سعی کنین بعد از خوندن هر بخش، سورت‌هایی که یاد گرفتین رو بررسی کنین و ببینین کدوم ویژگی رو دارن.

الگوریتم‌های مرتب‌سازی پایدار (Stable)

الگوریتم‌های پایدار یا همون Stable داده‌ها رو جوری سورت می‌کنن که ترتیب داده‌های یکسان به‌هم نخوره. این یعنی چی؟ تا حالا با سورت کردن آرایه‌های ساده مثل اعداد کار کردین و مهم نبود تو مسیر سورت چه اتفاقی برای داده‌ها می‌افته. ولی فرض کنین قراره امتیاز های یه سری دانشجو تو یه رقابت رو سورت کنین، که سرعت دانشجوها هم مهمه. اگه دو تا دانشجو امتیاز یکسان داشته باشن، دانشجویی که سریع‌تر بوده باید اول بیاد. مثلاً اگه آرایه ورودی $[4, 10, 23, 10, 15]$ باشه (که 10ها سرعت متفاوتی دارن)، تو آرایه خروجی ($[4, 10, 10, 15, 23]$) ترتیب 10ها باید همون ترتیب اولیه باشه.

از سورت‌هایی که دیدیم، اینسرشن سورت و مرج سورت پایدارن، ولی حبابی و انتخابی معمولاً پایدار نیستن، مگر اینکه تغییراتی توشون بدیم.

الگوریتم‌های مرتب‌سازی درجا (In-place)

تا حالا کلی درباره پیچیدگی زمانی حرف زدیم، حالا نوبت پیچیدگی مکانیه!

همون‌طور که پیچیدگی زمانی رو برای مقایسه سرعت الگوریتم‌ها تعریف کردیم، یه مفهوم به اسم پیچیدگی مکانی داریم که نشون می‌ده الگوریتم چقدر حافظه لازم داره. لازم نیست خیلی وارد جزئیات بشیم، تو درس ساختمان داده کامل باهاش آشنا می‌شین. ولی وقتی می‌گیم پیچیدگی مکانی یه الگوریتم $O(1)$ هست، یعنی به‌جز فضای داده‌های ورودی، حافظه اضافی که استفاده می‌کنیم یه ضریب منطقی از 1ه.

الگوریتم‌های In-place اونایی هستن که پیچیدگی مکانی $O(1)$ دارن. یعنی به‌جز آرایه اصلی، فقط از چند تا متغیر کمکی استفاده می‌کنن و مرتب‌سازی رو روی خود آرایه انجام می‌دن. از سورت‌هایی که

دیدیم، حبابی، انتخابی، و اینسرشن سورت In-place هستن، ولی مرج سورت چون آرایه‌های موقت می‌سازه، In-placeنیست.

الگوریتم‌های تطبیقی (Adaptive)

بعضی سورت‌ها بدون توجه به ترتیب داده‌های ورودی، همیشه یه تعداد عملیات ثابت دارن. ولی یه سری سورت‌ها برای داده‌هایی که تقریباً یا کاملاً مرتبن بهتر عمل می‌کنن؛ به اینا می‌گن سورت‌های تطبیقی.

مثلاً بیایین پیچیدگی زمانی اینسرشن سورت رو برای یه آرایه مرتب بررسی کنیم. این الگوریتم هر بار که می‌خواد یه عنصر جدید رو تو بخش مرتب‌شده جا بده، با یه مقایسه می‌فهمه که عنصر همون‌جاست و نیازی به جابه‌جایی نداره. حلقه‌ای که تو حالت عادی ممکنه O(n) عملیات داشته باشه، اینجا O(1) می‌شه. پس پیچیدگی کل می‌شه.O(n) × O(1) = O(n)

ما همچنان پیچیدگی اینسرشن سورت رو O(n^2) اعلام می‌کنیم، چون این کران بالاست، ولی حالا می‌دونیم برای آرایه‌های تقریباً مرتب، خیلی بهتر عمل می‌کنه. از سورت‌های ما، اینسرشن سورت تطبیقیه، ولی حبابی، انتخابی، و مرج سورت تطبیقی نیستن.

الگوریتم‌های مقایسه‌ای(Comparison)

سورت‌های مقایسه‌ای با مقایسه دوبه‌دو داده‌ها کار می‌کنن. فقط کافیه عناصر یه رابطه ترتیب تام داشته باشن (مثل اعداد یا رشته‌ها) تا این سورت‌ها بتونن مرتبشون کنن. همه سورت‌هایی که دیدیم (حبابی، انتخابی، درجی، ادغامی) مقایسه‌ای هستن. اگه بخواین آبجکت‌های پیچیده‌تر رو سورت کنین، کافیه کلاستون اینترفیس Comparable رو پیاده‌سازی کنه.

یه نکته جالب اینه که ثابت شده پیچیدگی زمانی سورت‌های مقایسه‌ای نمی‌تونه از O(n log n) بهتر باشه. یعنی هیچ‌جوری نمی‌تونین با مقایسه دوبه‌دو، سریع‌تر از این مرتب کنین! مرج سورت به این حد نزدیک می‌شه، ولی بقیه سورت‌هامون کندترن.

سورت‌های غیرمقایسه‌ای برای حالت‌های خاص طراحی شدن. اگه اطلاعات بیشتری درباره داده‌ها داشته باشین (مثلاً بدونین اعدادهاتون تو چه بازه‌ای هستن)، می‌تونین با پیچیدگی بهتر، مثلاًO(n) ،

مرتب‌سازی کنین. تو این داکیومنت درباره اینا حرف نمی‌زنیم، ولی اگه کنجکاوینRadix Sort و Counting Sortرو چک کنین.

چه چیزی یاد گرفتیم؟:

- الگوریتم‌ها چی‌ان و چرا تحلیل کاراییشون، به‌خصوص سرعت، مهمه .
- پیچیدگی زمانی چیه و چطور با نماد O (بیگO) می‌تونیم کارایی الگوریتم‌ها رو بدون وابستگی به سخت‌افزار و زبان برنامه‌نویسی مقایسه کنیم .
- اندازه ورودی تو تحلیل الگوریتم‌ها یعنی چی .
- ویژگی‌های مختلف الگوریتم‌های مرتب‌سازی مثل پایداری(Stable) ، مرتب‌سازی درجا(In-place)، پیچیدگی فضایی، و تطبیقی (Adaptive) بودن چه معنایی دارن .
- مرتب‌سازی‌های مقایسه‌ای (Comparison Sorts) چی‌ان، چه محدودیت (حد پایین O(n log n)) دارن، و با مرتب‌سازی‌های غیرمقایسه‌ای چه فرقی دارن .
- با چند الگوریتم مرتب‌سازی معروف آشنا شدیم، نحوه کارشون رو فهمیدیم و پیچیدگی زمانیشون رو تحلیل کردیم :
  - مرتب‌سازی حبابی (Bubble Sort) با پیچیدگی $O(n^2)$.
  - مرتب‌سازی انتخابی (Selection Sort) با پیچیدگی $O(n^2)$.
  - مرتب‌سازی درجی (Insertion Sort) با پیچیدگی $O(n^2)$ در حالت کلی، اما O(n) تو بهترین حالت (آرایه مرتب) و کارایی خوب برای داده‌های تقریباً مرتب.
  - مرتب‌سازی ادغامی (Merge Sort) با پیچیدگی O(n log n) که از روش تقسیم و حل استفاده می‌کنه.