طراحی الگوریتم

جزوه اول - تقسیم و حل

تقسیم و حل یک روش الگوریتمی است که برای حل مسائل پیچیده به کار می‌رود. در این روش، مسئله به زیرمسائل کوچکتر و مستقلی تقسیم می‌شود. هر زیرمسئله به‌صورت بازگشتی حل می‌شود و سپس نتایج این زیرمسائل با یکدیگر ترکیب می‌شوند تا به حل مسئله اصلی برسیم. مراحل کلی تقسیم و حل به این شکل است:

۱. **تقسیم:** مسئله را به چندین زیرمسئله مستقل که مشابه مسئله اصلی هستند، تقسیم کنید.

۲. **حل:** هر زیرمسئله را بازگشتی حل کنید. اگر زیرمسئله به اندازه کافی کوچک باشد، آن را مستقیما حل کنید.

۳. **ترکیب:** نتایج به‌دست آمده از حل زیرمسائل را ترکیب کنید تا جواب نهایی مسئله اصلی به دست آید.

از مهم‌ترین مثال‌های این روش Merge Sort و Binary Search هستند.

۱. Master Theorem

Master Theorem یک ابزار قدرتمند برای تحلیل الگوریتم‌های بازگشتی است. این قضیه به ما کمک می‌کند تا پیچیدگی زمانی بازگشتی را با استفاده از یک فرم کلی و ساده به دست آوریم. فرم کلی بازگشتی به صورت زیر است:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^c)$$

در این رابطه:

- a تعداد زیرمسائل است.

- $\frac{n}{b}$ اندازه هر زیرمسئله است.

- $O(n^c)$ هزینه‌ی ترکیب نتایج زیرمسائل است.

با استفاده از Master Theorem می‌توانیم سه حالت مختلف برای پیچیدگی زمانی را تعیین کنیم: (با فرض اینکه $d = \log_b a$)

۱. اگر $d > c$ باشد، زمان اجرای الگوریتم به صورت زیر است:

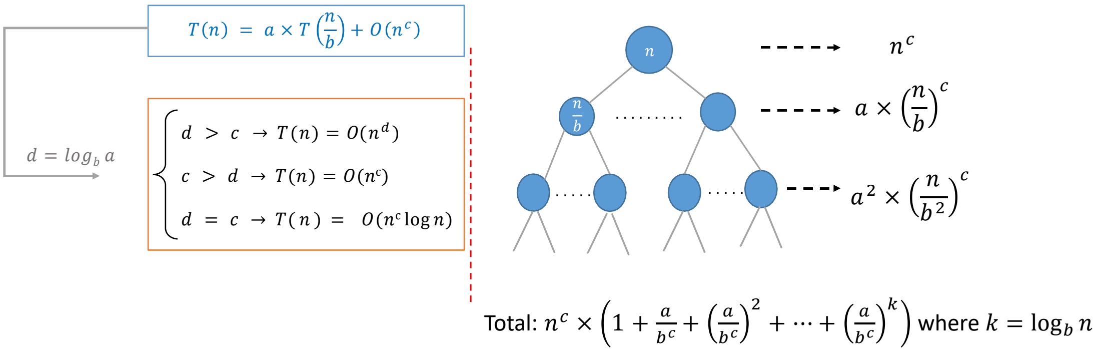
$$T(n) = O(n^d)$$

۲. اگر $c > d$ باشد، زمان اجرای الگوریتم به صورت زیر است:

$$T(n) = O(n^c)$$

۳. اگر $c = d$ باشد، زمان اجرای الگوریتم به صورت زیر است:

$$T(n) = O(n^c \log n)$$

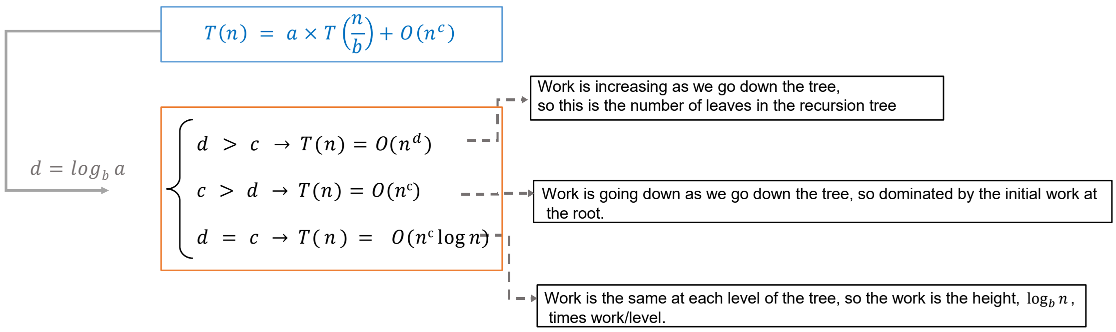


توضیح حالت‌ها:

۱. در حالت اول، $d > c$ به این معنی است که تعداد زیرمسائل (a) بیشتر از هزینه‌ی ترکیب (n^c) است. بنابراین، زمان کلی تحت تأثیر تعداد زیاد زیرمسائل قرار می‌گیرد و نتیجه نهایی به صورت $O(n^d)$ است.

۲. در حالت دوم، $c > d$ نشان‌دهنده این است که هزینه‌ی ترکیب بزرگ‌تر از تعداد زیرمسائل است. بنابراین، پیچیدگی زمانی به هزینه‌ی ترکیب یعنی $O(n^c)$ وابسته است.

۳. در حالت سوم، $c = d$ است و این حالت نشان می‌دهد که تأثیر هزینه‌ی ترکیب و تعداد زیرمسائل با یکدیگر برابر هستند. در این صورت، یک عامل اضافی $\log n$ به پیچیدگی زمانی اضافه می‌شود و نتیجه نهایی $O(n^c \log n)$ خواهد بود.



محدودیت‌ها:

Master Theorem تنها زمانی کاربرد دارد که ساختار بازگشتی مسئله دقیقا مطابق فرم زیر باشد:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^c)$$

در صورتی که هزینه‌ی تقسیم یا ترکیب به‌طور قابل‌توجهی پیچیده‌تر باشد یا ساختار بازگشتی متفاوت باشد، نمی‌توان از این قضیه استفاده کرد.

اثبات:

برای اثبات قضیه مستر، ابتدا رابطه بازگشتی زیر را در نظر می‌گیریم:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^c)$$

این رابطه بیان می‌کند که مسئله اصلی به a زیرمسئله با اندازه $\frac{n}{b}$ تقسیم می‌شود و هزینه ترکیب نتایج زیرمسائل $O(n^c)$ است.

حال، برای تحلیل این رابطه، تعداد سطوح مختلف در درخت بازگشتی و هزینه‌ی محاسبه در هر سطح را مورد بررسی قرار می‌دهیم.

۱. تعداد سطوح درخت:

در هر مرحله، مسئله به a زیرمسئله تقسیم می‌شود و اندازه هر زیرمسئله $\frac{n}{b}$ است. تعداد سطوح درخت بازگشتی برابر با تعداد مراحلی است که مسئله به اندازه کافی کوچک شود. به عبارت دیگر، تعداد مراحل تقسیم مسئله تا زمانی که اندازه هر زیرمسئله برابر ۱ شود، برابر است با:

$$\text{تعداد سطوح} = \log_b n$$

۲. هزینه هر سطح:

در هر سطح از درخت بازگشتی، تعداد زیرمسائل برابر a^k است (که k شماره سطح است). اندازه هر زیرمسئله در سطح k برابر $\frac{n}{b^k}$ است. بنابراین، هزینه محاسبه در هر سطح برابر است با:

$$\text{هزینه در سطح} k = a^k \cdot \left(\frac{n}{b^k}\right)^c = a^k \cdot \frac{n^c}{b^{kc}}$$

با توجه به اینکه $a = b^d$، می‌توانیم رابطه بالا را به صورت ساده‌تری بیان کنیم:

$$\text{هزینه در سطح} k = n^c \cdot \left(\frac{a}{b^d}\right)^k = n^c$$

بنابراین، هزینه کل مسئله برابر مجموع هزینه‌ها در تمامی سطوح است.

۳. جمع‌بندی حالت‌ها:

اکنون سه حالت زیر را با توجه به مقدار d و c بررسی می‌کنیم:

- **حالت اول :** اگر $d > c$ باشد، هزینه محاسبه در بالاترین سطح از درخت بازگشتی بیشترین تاثیر را دارد و هزینه کلی مسئله برابر است با:

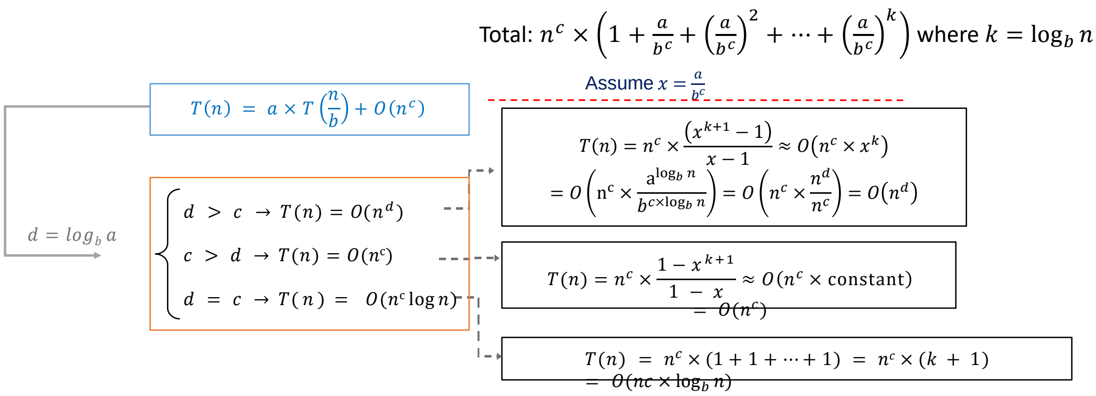
$$T(n) = O(n^d)$$

- **حالت دوم :** اگر $c > d$ باشد، هزینه ترکیب در هر سطح بیشتر از هزینه زیرمسائل است و پیچیدگی زمانی کل توسط هزینه ترکیب کنترل می‌شود:

$$T(n) = O(n^c)$$

- **حالت سوم :** اگر $d = c$ باشد، هزینه ترکیب و تعداد زیرمسائل تأثیر یکسانی دارند و یک عامل اضافی $\log n$ به هزینه نهایی اضافه می‌شود:

$$T(n) = O(n^c \log n)$$



مثال :

یک مثال ساده از این قضیه، تحلیل الگوریتم Merge Sort است که رابطه بازگشتی آن به‌صورت زیر است:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

در اینجا $a = 2$، $b = 2$ و $c = 1$ است. با توجه به اینکه $d = \log_2 2 = 1$ و $d = c$، این مسئله در حالت سوم قضیه مستر قرار می‌گیرد. بنابراین پیچیدگی زمانی الگوریتم به صورت $O(n \log n)$ خواهد بود.

در اینجا $a = 2$، $b = 2$ و $d = 1$ است. با توجه به اینکه $a = b^d$ است (زیرا $2 = 2^1$)، این مسئله در حالت دوم قضیه مستر قرار می‌گیرد و پیچیدگی زمانی آن به صورت $O(n \log n)$ خواهد بود.

۲. محاسبه a^n

می‌خواهیم حاصل a^n را بدست آوریم. برای این کار بدیهی‌ترین راهی که به ذهن میرسید این است که از یک حلقه و ضرب متوالی استفاده کرد.

Algorithm ۱ Power Calculation (Iterative)

```
function power(a, n)
    result ← 1
    for i = 1 to n do
        result ← result × a
    end for
    return result
end function
```

اما این روش بهینه ای نیست و باید راه حل بهتری برای آن پیدا کنیم. اگر بخواهیم از روش بازگشتی استفاده کنیم اولین راه حلی که به ذهن میرسید ممکن است چیزی شبیه کد زیر باشد.

Algorithm ۲ Power Calculation (Recursive)

```
function power(a, n)
    if n == 0 then
        return 1
    end if
    return power(a, n − 1) × a
end function
```

اما این کد نیز پیچیدگی $O(n)$ دارد که برای ما مناسب نیست. (این کد به دلیل اینکه باید از stack های زیادی و function call های متعدد پشتیبانی کند در عمل از کد اول کندتر است)

حال سعی میکنیم الگوریتم را کمی بهتر کنیم و سعی میکنیم به شکل بهتری مسئله را بشکنیم:

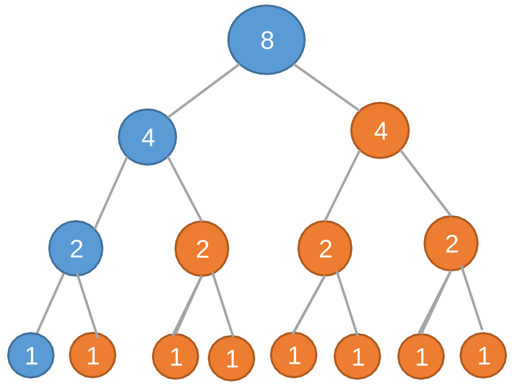
Algorithm ۳ Power Calculation (Optimized Recursive)

```
function power(a, n)
    if n == 0 then
        return 1
    else if n == 1 then
        return a
    end if
    return power(a, ⌊n/2⌋) × power(a, ⌈n/2⌉)
end function
```

به نظر شما مشکل کد بالا چیست؟ سعی کنید قبل از خواندن بقیه مطلب جواب این سوال را بدهید.

مشکل کد این است که در این کد بعضی از محاسبات چندین بار تکرار می‌شوند و عملا بهینه‌سازی‌ای انجام نشده.

$$T(n) = 2T(n/2) + O(1) \approx O(n)$$

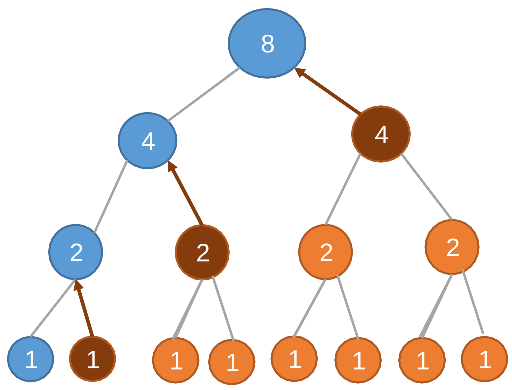


در شکل بالا نودهایی که با رنگ نارنجی مشخص شده‌اند نیاز به محاسبه نداشتند و می‌توانستیم با ذخیره نتایج قبلی از دوباره کاری جلوگیری کنیم.

برای اینکه از محاسبه دوباره جلوگیری کنیم باید از روشی به نام Memoization استفاده کنیم. Memoization به معنی ذخیره کردن (caching) نتایج بخش‌های پرهزینه کد می‌باشد.

Algorithm ۴ top down

function power(a, n)
    if $n == 0$ **then**
        return 1
    end if
    if $A[n] \neq$ null **then**
        return $A[n]$
    end if
    $A[n] \leftarrow \text{power}(a, \lfloor n/2 \rfloor) \times \text{power}(a, \lceil n/2 \rceil)$
    if $n\%2 == 1$ **then**
        $A[n] \leftarrow A[n] \times a$
    end if
    return $A[n]$
end function



در پایتون نیز روش ساده‌تری برای این کار وجود دارد:

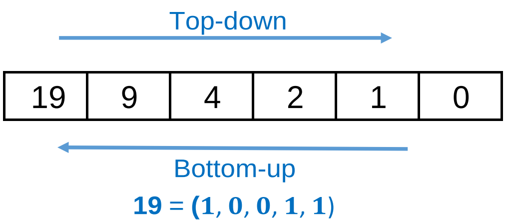
```python
@lru_cache(None)
def power(a, n):
    if n == 0:
        return 1
    if n == 1:
        return a

    return power(a, floor(n / 2)) * power(a, ceil(n / 2))
```

در مورد این موضوع می‌توانید اینجا مطالعه بیشتری داشته باشید.

همچنین میتوانیم این مسئله را با منطق فکری bottom-up نیز حل کنیم. در حالت قبلی ما مسئله بزرگ خود را می‌شکستیم و مسئله‌های کوچک‌تر را با کمک آن حل میکردیم اما می‌توانیم از مسئله‌های کوچک‌تر نیز شروع کنیم و مسئله بزرگ را با آن‌ها حل کنیم.

برای این کار می‌توانیم از نمایش بر مبنای دو توان استفاده کنیم و از کم‌ارزش‌ترین بیت شروع کنیم و هر دفعه مقدار را آپدیت کنیم.

شمای کلی این راه‌حل در مقابل راه‌حل قبلی به شکل زیر می‌باشد:



کد زیر پیاده‌سازی این الگوریتم را نشان می‌دهد. سعی کنید در مورد عملکرد آن فکر کنید.

Algorithm ۵ bottom up

if $n = 0$ **then**
  return 1
end if
$n\_binary\_repr \leftarrow [n_k, n_{k-1}, \dots, n_0]$
$result \leftarrow a$
for $i = k - 1$ to 0 **do**
  if $n[i] = 0$ **then**
    $result \leftarrow result \times result$
  else
    $result \leftarrow result \times result \times a$
  end if
end for
return $result$

از آنجایی که این الگوریتم به اندازه طول عدد در مبنای 2 تکرار می‌شود، پس پیچیدگی زمانی آن برابر است با $O(\log n)$ و پیچیدگی مکانی آن نیز برابر است با $O(\log n)$ چون یک آرایه به این اندازه نیاز داریم.

۳. Polynomial Multiplication

مسئله ضرب چندجمله‌ای‌ها به این صورت است که دو چندجمله‌ای $A(x)$ و $B(x)$ از درجه n داده شده است و هدف محاسبه $C(x)$ است که حاصل‌ضرب $A(x)$ و $B(x)$ می‌باشد.

$$A(x) = \sum_{i=0}^{n} a_i x^i, \quad B(x) = \sum_{i=0}^{n} b_i x^i$$

$$C(x) = A(x) \times B(x) = \sum_{i=0}^{2n} c_i x^i$$

که در آن هر ضریب c_k برابر است با مجموع حاصل‌ضرب‌های a_i و b_j که مجموع اندیس‌های آن‌ها برابر با k است:

$$c_k = \sum_{i=0}^{k} a_i \cdot b_{k-i}$$

روش ساده (پیچیدگی $O(n^2)$)

روش ساده محاسبه ضرب دو چندجمله‌ای این است که هر ضریب c_k را با استفاده از رابطه بالا محاسبه کنیم. در این روش برای هر k، c_k از مجموع n حاصل‌ضرب تشکیل شده است و در نتیجه کل پیچیدگی زمانی این روش $O(n^2)$ خواهد بود.

$$c_k = a_0 \cdot b_k + a_1 \cdot b_{k-1} + \cdots + a_k \cdot b_0$$

روش تقسیم و حل (پیچیدگی $O(n^{\log_2 3})$)

برای بهینه‌سازی این روش، از تکنیک ''تقسیم و حل'' استفاده می‌کنیم. در این روش، چندجمله‌ای‌ها را به دو بخش تقسیم می‌کنیم:

$$A(x) = A_0(x) + A_1(x) \cdot x^{n/2}, \quad B(x) = B_0(x) + B_1(x) \cdot x^{n/2}$$

که در آن $A_0(x)$ و $B_0(x)$ شامل جمله‌های قسمت پایین چندجمله‌ای و $A_1(x)$ و $B_1(x)$ شامل جمله‌های قسمت بالای چندجمله‌ای هستند. سپس حاصل‌ضرب $A(x) \times B(x)$ به صورت زیر به چهار زیرمسئله تقسیم می‌شود:

$$A(x) \times B(x) = A_0(x) \times B_0(x) + A_0(x) \times B_1(x) \cdot x^{n/2} + A_1(x) \times B_0(x) \cdot x^{n/2} + A_1(x) \times B_1(x) \cdot x^n$$

حال برای کاهش تعداد زیرمسائل، از روش زیر استفاده میکنیم. ایده اصلی این روش این است که به جای محاسبه جداگانه حاصل‌ضرب $A_0(x) \times B_1(x)$ و $A_1(x) \times B_0(x)$، از رابطه زیر استفاده کنیم:

$$A_0(x) \times B_1(x) + A_1(x) \times B_0(x) = (A_0(x) + A_1(x)) \times (B_0(x) + B_1(x)) - A_0(x) \times B_0(x) - A_1(x) \times B_1(x)$$

این کار باعث می‌شود که به جای چهار زیرمسئله تنها سه زیرمسئله داشته باشیم:

۱. $A_0(x) \times B_0(x)$

۲. $A_1(x) \times B_1(x)$

۳. $(A_0(x) + A_1(x)) \times (B_0(x) + B_1(x))$

در نتیجه پیچیدگی زمانی این روش به $O(n^{\log_2 3})$ کاهش پیدا می‌کند که تقریباً برابر با $O(n^{1.58})$ است.

شبه‌کد روش تقسیم و حل برای ضرب چندجمله‌ای‌ها:

Algorithm ۶ Polynomial Multiplication using Divide and Conquer

```
function poly_multiply(A, B, n)
    if n == 1 then
        return A[0] × B[0]
    end if
    Divide A into A_0 and A_1
    Divide B into B_0 and B_1
    P_0 ← poly_multiply(A_0, B_0, n/2)
    P_1 ← poly_multiply(A_1, B_1, n/2)
    S_A ← A_0 + A_1
    S_B ← B_0 + B_1
    P_2 ← poly_multiply(S_A, S_B, n/2)
    P_2 ← P_2 − P_0 − P_1
    return Combine P_0, P_1, P_2 using coefficients x^{n/2} and x^n
end function
```

تحلیل پیچیدگی زمانی

پیچیدگی زمانی روش ساده $O(n^2)$ است. اما با استفاده از روش کاراتسوبا و تقسیم به سه زیرمسئله، پیچیدگی زمانی به $O(n^{\log_2 3})$ که تقریباً برابر $O(n^{1.58})$ است کاهش پیدا می‌کند. این روش نسبت به روش ساده بسیار سریع‌تر است، به‌ویژه برای چندجمله‌ای‌هایی با درجه بالا.

۴. Maximum Subarray Sum

در این مسئله، شما یک آرایه از اعداد صحیح داده شده دارید که شامل اعداد مثبت و منفی است. هدف این است که مجموع بزرگترین زیرآرایه پیوسته را پیدا کنید.

ورودی: یک آرایه A شامل n عدد صحیح.

خروجی: زیرآرایه‌ای که مجموع اعداد آن بیشترین مقدار ممکن است.

راه حل $O(n^2)$

یک روش ساده برای حل این مسئله بررسی تمام زیرآرایه‌های ممکن است. می‌توانیم با دو حلقه تو در تو تمامی زیرآرایه‌های ممکن را بررسی کنیم و مجموع آن‌ها را محاسبه کنیم.

Algorithm ۷ $O(n^2)$ - Maximum Subarray

```
max_sum ← −∞
for i = 0 to n − 1 do
    sum ← 0
    for j = i to n − 1 do
        sum ← sum + A[j]
        if sum > max_sum then
            max_sum ← sum
        end if
    end for
end for
return max_sum
```

تحلیل پیچیدگی زمانی

پیچیدگی زمانی این الگوریتم $O(n^2)$ است. چون برای هر انتخاب i از 0 تا $n-1$، یک حلقه دیگر برای j از i تا $n-1$ اجرا می‌شود. به همین دلیل دو حلقه تو در تو منجر به پیچیدگی $O(n^2)$ می‌شود.

روش تقسیم و حل $O(n \log n)$

روش بهینه‌تر برای حل این مسئله استفاده از تکنیک تقسیم و حل است. در این روش آرایه به دو بخش تقسیم می‌شود و سپس نتایج برای زیرآرایه‌های چپ و راست محاسبه و ترکیب می‌شوند.

مراحل حل مسئله

۱. **تقسیم (Divide):** آرایه را به دو نیمه تقسیم می‌کنیم. به این صورت که:

$$\text{mid} = \left\lfloor \frac{\text{first} + \text{last}}{2} \right\rfloor$$

سپس آرایه به دو زیرآرایه $[first, mid]$ و $[mid+1, last]$ تقسیم می‌شود.

۲. **حل زیرمسائل (Conquer):** برای هر نیمه، مسئله به صورت بازگشتی حل می‌شود. یعنی:

- مجموع زیرآرایه‌ای با بزرگترین مقدار در زیرآرایه چپ را پیدا می‌کنیم.
- مجموع زیرآرایه‌ای با بزرگترین مقدار در زیرآرایه راست را پیدا می‌کنیم.

۳. **ترکیب (Combine):** پس از حل زیرمسائل، نتایج به این صورت ترکیب می‌شوند: بزرگترین مجموع زیرآرایه‌ای که از بخش چپ شروع شده و به بخش راست می‌رسد را پیدا می‌کنیم. برای این کار، از المنت‌های میانی به سمت چپ حرکت کرده و بیشترین مجموع را محاسبه می‌کنیم:

$$\text{left\_sum} = \max\left(\sum_{i=\text{mid}}^{\text{first}} A[i]\right)$$

سپس برای زیرآرایه راست نیز همین کار را انجام می‌دهیم:

$$\text{right\_sum} = \max\left(\sum_{i=\text{mid}+۱}^{\text{last}} A[i]\right)$$

در نهایت:

$$\text{cross\_sum} = \text{left\_sum} + \text{right\_sum}$$

بزرگترین مجموع نهایی برابر است با بیشترین مقدار بین سه حالت زیر:

$$\text{max\_sum} = \max(\text{left\_max}, \text{right\_max}, \text{cross\_sum})$$

تحلیل پیچیدگی زمانی

پیچیدگی زمانی این الگوریتم $O(n \log n)$ است. در هر مرحله آرایه به دو بخش تقسیم شده و هر بار محاسباتی با پیچیدگی $O(n)$ انجام می‌شود.

شبه‌کد

Algorithm ۸ $O(n \log n)$ - Maximum Subarray

```
function find_max_subarray(A، first، last)
    if first == last then
        return A[first]
    end if
    mid ← ⌊(first+last)/2⌋
    left_max ← find_max_subarray(A، first، mid)
    right_max ← find_max_subarray(A، mid + 1، last)
    left_sum ← −∞
    sum ← 0
    for i = mid downto first do
        sum ← sum + A[i]
        if sum > left_sum then
            left_sum ← sum
        end if
    end for
    right_sum ← −∞
    sum ← 0
    for i = mid + 1 to last do
        sum ← sum + A[i]
        if sum > right_sum then
            right_sum ← sum
        end if
    end for
    cross_sum ← left_sum + right_sum
    return max(left_max, right_max, cross_sum)
end function
```

می‌توان برای این بخش مانند اسلایدهای درس نیز عمل کرد و یک آرایه کمکی B تعریف کرد و از آن استفاده کرد تا پیچیدگی زمانی را به $O(n)$ کاهش داد.

روش کادان $O(n)$

روش دیگری که می‌توان برای حل مسئله استفاده کرد، الگوریتم کادان (Kadane's Algorithm) است. این الگوریتم با یک گذر ساده از آرایه، بزرگترین مجموع زیرآرایه را محاسبه می‌کند.

شبه‌کد

Algorithm ۹ کادان الگوریتم $O(n)$

$max\_so\_far \leftarrow A[0]$
$max\_ending\_here \leftarrow A[0]$
for $i = 1$ to $n - 1$ **do**
    $max\_ending\_here \leftarrow \max(A[i], max\_ending\_here + A[i])$
    $max\_so\_far \leftarrow \max(max\_so\_far, max\_ending\_here)$
end for
return max_so_far

تحلیل پیچیدگی زمانی

پیچیدگی زمانی این الگوریتم $O(n)$ است، زیرا تنها یک گذر از آرایه انجام می‌شود.

۵. Closest Pair of Points

مسئله نزدیک‌ترین جفت نقاط به این صورت است که مجموعه‌ای از n نقطه در صفحه دوبعدی داده شده است و هدف یافتن دو نقطه‌ای است که کمترین فاصله را از یکدیگر دارند. حل این مسئله به روش ساده و $O(n^2)$ بسیار وقت‌گیر است، چرا که باید فاصله تمام جفت نقاط را محاسبه کنیم و کمترین آن را پیدا کنیم. اما با استفاده از روش ”تقسیم و حل“ می‌توان این مسئله را در زمان $O(n \log n)$ حل کرد.

مراحل حل مسئله با روش تقسیم و حل

این روش به سه مرحله اصلی تقسیم می‌شود: تقسیم (Divide)، حل زیرمسئله‌ها (Conquer) و ترکیب (Combine).

۱. تقسیم (Divide): ابتدا نقاط را بر اساس مختصات x مرتب می‌کنیم. سپس مجموعه نقاط را به دو نیمه تقسیم می‌کنیم؛ نیمه چپ شامل نقاط با مختصات x کمتر از مقدار میانه و نیمه راست شامل نقاط با مختصات x بیشتر از مقدار میانه. این کار با محاسبه نقطه میانه M انجام می‌شود:

$$M = \mathrm{P}[n/2]$$

سپس مجموعه نقاط به دو زیرمجموعه تقسیم می‌شود: مجموعه P_L (نقاط چپ) و مجموعه P_R (نقاط راست).

۲. حل زیرمسئله‌ها (Conquer): حال برای هر نیمه به صورت بازگشتی مسئله را حل می‌کنیم: - نزدیک‌ترین جفت نقاط در نیمه چپ d_L - نزدیک‌ترین جفت نقاط در نیمه راست d_R

حال کوچکترین فاصله را بین دو زیرمسئله داریم:

$$d = \min(d_L, d_R)$$

۳. ترکیب (Combine): در نهایت باید بررسی کنیم که آیا ممکن است نزدیک‌ترین جفت نقاط یک نقطه از نیمه چپ و یک نقطه از نیمه راست باشند. برای این کار، باید تنها نقاطی را در نظر بگیریم که در یک نوار عمودی به پهنای $2d$ در اطراف خط میانه قرار دارند (یعنی نقاطی که مختصات x آن‌ها بین $M - d$ و $M + d$ قرار دارد).

برای این کار ابتدا مجموعه‌ای از نقاطی که در این نوار عمودی قرار دارند (به نام P_{strip}) تشکیل می‌دهیم. سپس این نقاط را بر اساس مختصات y مرتب می‌کنیم.

حال برای هر نقطه در P_{strip}، باید فاصله آن را با چند نقطه بعدی که در محدوده d قرار دارند بررسی کنیم (حداکثر ۷ نقطه بعدی). این کار با توجه به اینکه نقاط بر اساس y مرتب شده‌اند، انجام می‌شود. در نهایت کوچکترین فاصله‌ای که در این بررسی‌ها یافت می‌شود، فاصله نهایی خواهد بود.

شبه‌کد:

Algorithm ۱۰ Closest Pair of Points using Divide and Conquer

```
function closest_pair(P)
    if n ≤ 3 then
        return the closest pair by direct comparison
    end if
    Sort the points in P based on their x-coordinates
    mid ← n/2
    P_L ← P[1 ... mid]
    P_R ← P[mid + 1 ... n]
    d_L ← closest_pair(P_L)                ▷ Find closest pair in the left half
    d_R ← closest_pair(P_R)                ▷ Find closest pair in the right half
    d ← min(d_L, d_R)                      ▷ Find the minimum distance between left and right halves
    P_strip ← points in P where x-coordinate is within [M − d, M + d]
    Sort the points in P_strip by their y-coordinates
    for each point p_i in P_strip do
        for each point p_j in P_strip where j ≤ i + 7 do
            calculate the distance between p_i and p_j
            d ← min(d, distance(p_i, p_j))
        end for
    end for
    return d
end function
```

البته شبه‌کد بالا مشکل دارد و مرتبه زمانی آن چیزی نیست که دنبالش بودیم. به نظر شما دلیل این موضوع چیست؟

بهینه‌سازی شبه‌کد و تحلیل زمانی

یکی از مشکلات اصلی در پیاده‌سازی اولیه این الگوریتم، عدم توجه به هزینه مرتب‌سازی نقاط است. به همین دلیل، باید توجه کنیم که مرتب‌سازی نقاط بر اساس مختصات x و y به روش مناسبی انجام شود.

۱. پیش‌پردازش (Preprocess):

ابتدا نقاط را بر اساس مختصات مرتب می‌کنیم که این کار در زمان $O(n \log n)$ انجام می‌شود. سه آرایه مختلف را نگهداری می‌کنیم:

- P: آرایه نقاط اصلی
- X: اندیس نقاط با فرض مرتب‌سازی بر اساس مختصات x
- Y: اندیس نقاط با فرض مرتب‌سازی بر اساس مختصات y

ایجاد آرایه‌های X و Y در زمان $O(n \log n)$ انجام می‌شود.

۲. تقسیم :(Divide)

در این مرحله، نقاط را به دو نیمه تقسیم می‌کنیم: نیمه چپ و نیمه راست، با استفاده از مقدار میانه M. برای این کار، به نقاط مرتب شده بر اساس مختصات x نیاز داریم که از قبل داریم. سپس با استفاده از مختصات میانه M که برابر با مقدار مختصات x نقطه میانی در آرایه مرتب‌شده است، نقاط را به دو مجموعه چپ و راست تقسیم می‌کنیم.

علاوه بر این، از آرایه Y که نقاط را بر اساس مختصات y مرتب کرده است، استفاده می‌کنیم تا به‌طور کارآمد مجموعه‌های Y_L و Y_R را ایجاد کنیم. این آرایه‌ها به ما کمک می‌کنند تا در مراحل بعدی الگوریتم، ترکیب (Com-bine) بهینه‌تر انجام شود.

شبه‌کد نهایی برای بخش تقسیم:

Algorithm ۱۱ Updated Divide Step using Y array

```
Sort the points in P based on their x-coordinates        ▷ O(n log n) preprocessing
mid ← n/2
M ← P[X[mid]].x                                           ▷ Find the median based on sorted X array
Initialize empty arrays Y_L، Y_R
for each point P[Y[i]] in Y do
    if P[Y[i]].x < M then
        Add Y[i] to Y_L
    else
        Add Y[i] to Y_R
    end if
end for
```

تقسیم‌بندی بالا برای بعضی حالت‌های case corner اشکال دارد، سعی کنید این مشکل را حل کنید.

۳. حل زیرمسئله‌ها :(Conquer)

برای حل زیرمسئله‌ها به صورت بازگشتی، کوچکترین فاصله‌ها را در نیمه چپ و نیمه راست محاسبه می‌کنیم:

$$d = \min(d_L, d_R)$$

۴. ترکیب :(Combine)

برای بررسی این که آیا نزدیک‌ترین جفت نقاط ممکن است یکی در نیمه چپ و دیگری در نیمه راست باشد، نقاطی که در نوار عمودی به پهنای $2d$ قرار دارند را در نظر می‌گیریم. برای بهینه‌سازی این مرحله:

- ابتدا نقاط این نوار را مرتب‌سازی می‌کنیم (که از قبل در آرایه Y به صورت مرتب‌شده وجود دارد و صرفا نیاز داریم که با یک پیمایش خطی بر روی نقاط آن‌هایی که در محدوده مناسب نیستند را حذف کنیم).

- سپس هر نقطه را حداکثر با ۷ نقطه بعدی خود مقایسه می‌کنیم که این عملیات در زمان $O(n)$ انجام می‌شود.

شبه‌کد بهینه‌شده:

Algorithm ۱۲ Optimized Closest Pair of Points using Divide and Conquer

```
function closest_pair(P)
    if n ≤ 3 then
        return the closest pair by direct comparison
    end if
    Sort the points in P based on their x-coordinates        ▷ O(n log n) preprocessing
    mid ← n/2
    P_L ← P[1 ... mid]
    P_R ← P[mid + 1 ... n]
    d_L ← closest_pair(P_L)
    d_R ← closest_pair(P_R)
    d ← min(d_L, d_R)
    P_strip ← points in P where x-coordinate is within [M − d, M + d]
    Sort the points in P_strip by their y-coordinates        ▷ Already sorted
    for each point p_i in P_strip do
        for each point p_j in P_strip where j ≤ i + 7 do
            calculate the distance between p_i and p_j
            d ← min(d, distance(p_i, p_j))
        end for
    end for
    return d
end function
```

تحلیل پیچیدگی زمانی با توجه به این که مرحله پیش‌پردازش شامل مرتب‌سازی نقاط در زمان $O(n \log n)$ است و هر مرحله بازگشتی تقسیم و ترکیب نیز در زمان $O(n)$ انجام می‌شود، می‌توان پیچیدگی زمانی کلی الگوریتم را به صورت زیر تحلیل کرد:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \approx O(n \log n)$$

این نشان می‌دهد که با استفاده از روش ''تقسیم و حل''، پیچیدگی زمانی الگوریتم به $O(n \log n)$ کاهش یافته است که بسیار کارآمدتر از روش ساده $O(n^2)$ است.

۶. نقاش بدون امکانات

نقاشی می‌خواهد حصار خانه‌ای را رنگ‌آمیزی کند. تنها ابزاری که در اختیار دارد، یک قلم‌مو با عرض ۱ متر می‌باشد که با آن می‌تواند حصار را به صورت حرکات عمودی یا افقی رنگ کند. توجه داشته باشید در هر حرکت برای رنگ کردن حصار، تمامی سطح قلم باید حصار را لمس کند.
این حصار شامل n تخته‌ی عمودی است که در کنار هم چیده شده‌اند. بین تخته‌های مجاور هیچ فاصله‌ای وجود ندارد. تخته‌ها از چپ به راست شماره‌گذاری شده‌اند و تخته‌ی iام دارای عرض ۱ متر و ارتفاع a_i می‌باشد.
حداقل چند حرکت لازم است تا نقاش بتواند کل حصار را رنگ‌آمیزی کند؟ توجه داشته باشید که او می‌تواند هر ناحیه‌ای از حصار را چندین بار رنگ‌آمیزی کند.

پاسخ: برای حل این مسئله باید به چند نکته توجه کنیم. اول اینکه هر حرکت افقی باید تا حد امکان عریض باشد. دوم اینکه، زیر هر حرکت افقی باید فقط حرکات افقی وجود داشته باشد. بنابراین، اگر پایین حصار با یک حرکت

افقی رنگ‌آمیزی شده باشد، تعداد این ضربات باید حداقل برابر با $min(a_1, a_2, ..., a_n)$ باشد. این حرکات ممکن است حصار را به چند بخش رنگ‌نشده که از هم جدا هستند، تقسیم کنند. برای همه‌ی این بخش‌ها باید جواب‌های آن‌ها را جمع کنیم. اکنون متوجه می‌شویم که می‌توانیم همین روش را به صورت بازگشتی برای بخش‌های رنگ نشده، اعمال کنیم. حالت دیگری نیز وجود دارد که یک بخش را می‌توان تنها با حرکات عمودی نیز رنگ کرد و ما باید برای هر بخش تصمیم بگیریم که کدام روش بهینه است.
از آنجا که n تخته داریم و مقدار کمینه هر بخش را می‌توان با استفاده از *segment tree* در اردر $O(logn)$ یافت، اردر راه‌حل برابر $O(nlogn)$ می‌شود.

۷. منابع آموزش *segment tree*

۱. وبلاگ حسام حداد

این مقاله به زبان فارسی نوشته شده و اصول اولیه *segment tree* را به‌طور مفصل توضیح می‌دهد. ابتدا، مفاهیم پایه‌ای و ساختار *segment tree* معرفی می‌شوند. سپس، نحوه ساخت درخت، به‌روزرسانی داده‌ها، و جستجوی محدوده‌ای با استفاده از این ساختار تشریح می‌شود. در این وبلاگ، مثال‌های ساده و کاربردی برای هر یک از مفاهیم آورده شده است که می‌تواند به دانشجویان کمک کند تا مفاهیم را بهتر درک کنند و آن‌ها را در مسائل مختلف به کار بگیرند.
لینک: `https://hesamhaddad.blog.ir/1393/03/23/Segment-Tree`

۲. کتاب ساختار داده‌ی GTOI

این بخش از کتاب ساختار داده GTOI به زبان فارسی نوشته شده است و یکی از جامع‌ترین منابع برای یادگیری *segment tree* در قالب نوشتاری است. در این منبع علاوه بر پوشش اصول پایه‌ای، مطالبی چون بهینه‌سازی به‌روزرسانی‌ها و جستجوها و تکنیک‌های پیشرفته در کار با *segment tree* توضیح داده می‌شوند. همچنین، پیاده‌سازی‌های کد به زبان C++ ارائه شده است که می‌تواند به شما کمک کند تا به‌طور عملی نحوه کارکرد *segment tree* را ببینند و مهارت‌های خود را در این زمینه تقویت کنند.
لینک: `https://gtoi.shaazzz.ir/book/8/5.html`

۳. ویدئوی یوتیوب از Roy Tushar

این ویدئو به زبان انگلیسی تهیه شده و توسط Roy Tushar ارائه می‌شود. ویدئو به‌صورت گام‌به‌گام نحوه ساخت *segment tree* ، به‌روزرسانی داده‌ها و اجرای کوئری‌های محدوده‌ای را با مثال‌های کاربردی و کدنویسی عملی توضیح می‌دهد. در طول ویدئو، نحوه تفکر درباره مسائل و بهینه‌سازی الگوریتم‌ها در استفاده از *segment tree* نیز آموزش داده می‌شود. دانشجویان می‌توانند از این ویدئو برای درک عملی بهتر و کاربردی‌تر *segment tree* بهره‌مند شوند و آن را به عنوان یک منبع مکمل در کنار منابع متنی به کار گیرند.
لینک: `https://www.youtube.com/watch?v=ZBHKZF5w4YU`