

به نام خدا

طراحی الگوریتم‌ها

آرش شفیعی



برنامه ریزی پویا

- برنامه‌ریزی پویا¹ روشی دیگر برای حل مسائل محاسباتی است که توسط آن همانند روش تقسیم و حل جواب یک مسئله از جواب زیر مسئله‌های آن به دست می‌آید. (در اینجا واژه programming به معنی برنامه‌ریزی و طراحی یک جدول برای پیش‌بینی آینده است و نه به معنی برنامه نویسی).
- در برنامه‌ریزی پویا هر یک از زیر مسئله‌ها تنها یک بار حل می‌شوند. جواب یک زیرمسئله در یک جدول ذخیره می‌شود و از آن جواب برای حل زیر مسئله‌های دیگر با اندازه‌های بزرگتر استفاده می‌شود. در واقع در هر مرحله یک زیر مسئله بزرگ‌تر با استفاده از جواب یک زیر مسئله کوچک‌تر حل می‌شود و این روند ادامه پیدا می‌کند تا اینکه مسئله اصلی با استفاده از بزرگترین زیرمسئله به دست آمده حل می‌شود.
- برنامه‌ریزی پویا در بسیاری از مسائل بهینه‌سازی² کاربرد دارد. چنین مسئله‌هایی معمولاً چند جواب دارند که ما به دنبال جوابی می‌گردیم که مقدار بهینه (کوچکترین یا بزرگترین) داشته باشد.

¹ dynamic programming

² optimization problem

- برای مثال برای پیدا کردن عدد فیبوناچی n ام، باید عدد فیبوناچی $n-1$ ام و عدد فیبوناچی $n-2$ ام را محاسبه کنیم.
- می‌توانیم یک رابطه بازگشتی برای محاسبه عدد فیبوناچی بنویسیم و با استفاده از یک الگوریتم بازگشتی آن را حل کنیم. مشکل الگوریتم بازگشتی این است که برخی از زیرمسئله‌ها بیش از یک بار حل می‌شوند. برای مثال برای محاسبه عدد فیبوناچی پنجم عدد فیبوناچی چهارم و سوم محاسبه می‌شوند. اما هنگام محاسبه عدد فیبوناچی چهارم عدد سوم برای بار دوم باید محاسبه شود.

- یک روش برای حل این مشکل این است که به جای حل مسئله از بالا به پایین، یعنی با شروع از مسئله بزرگ‌تر و محاسبه زیرمسئله‌های کوچک‌تر، مسئله را از پایین به بالا حل کنیم، بدین معنی که ابتدا زیرمسئله را حل کنیم و نتایج را ذخیره کرده و از نتایج در مسئله‌های بزرگ‌تر استفاده کنیم.
- برای مثال در مسئله محاسبه عدد فیبوناچی n ام، ابتدا عدد فیبوناچی اول، سپس دوم، سوم، .. را محاسبه کرده تا به عدد فیبوناچی n ام برسیم.
- این روش حل مسئله از پایین به بالا با شروع به محاسبه زیرمسئله‌های کوچک‌تر و استفاده از جواب زیرمسئله‌ها در مسئله‌های بزرگ‌تر برنامه‌ریزی پویا نامیده می‌شود.

ضرب زنجیره‌ای ماتریس‌ها

- مسئله ضرب زنجیره‌ای ماتریس‌ها¹ به صورت زیر است. می‌خواهیم دنباله (زنجیره)ای از n ماتریس $\langle A_1, A_2, \dots, A_n \rangle$ را در هم ضرب کنیم. این ماتریس‌ها الزاماً ماتریس‌های مربعی نیستند و هدف این است که در این ضرب ماتریسی کمترین تعداد عملیات ضرب استفاده شود.
- ضرب ماتریس‌ها شرکت‌پذیر²، بدین معنی که پرانتزگذاری به هر نحوی انجام می‌شود، جواب ضرب ماتریسی تغییر نخواهد کرد.

¹ Matrix-chain multiplication problem

² associative

ضرب زنجیره‌ای ماتریس‌ها

- الگوریتم ضرب دو ماتریس $A(a_{ij})$ و $B(b_{ij})$ به صورت زیر است. نتیجه ضرب این دو ماتریس در ماتریس $C(c_{ij})$ ذخیره می‌شود.

Algorithm Matrix Multiplication

```
function RECTANGULAR-MATRIX-MULTIPLY(A, B, C, p, q, r)
1: for i = 1 to p do
2:   for j = 1 to r do
3:     for k = 1 to q do
4:       c[i,j] += a[i,k] * b[k,j]
```

- برای اینکه ضرب ماتریسی درست باشد لازم است ابعاد ماتریس A برابر با $p \times q$ و ابعاد ماتریس B برابر با $q \times r$ باشد و ابعاد ماتریس حاصل ضرب C در این صورت برابر با $p \times r$ خواهد بود. تعداد عملیات ضرب انجام شده برابر است با pqr .

ضرب زنجیره‌ای ماتریس‌ها

- زنجیره ضرب ماتریسی $A_1 \cdot A_2 \cdot A_3$ را در نظر بگیرید. فرض کنید ماتریس A_1 با ابعاد 10×100 ، ماتریس A_2 با ابعاد 100×5 و ماتریس A_3 با ابعاد 5×50 باشد. اگر پرانتز گذاری به صورت $((A_1 A_2) A_3)$ باشد، تعداد $10 \times 100 \times 5 = 5000$ عملیات ضرب برای ضرب $A_1 A_2$ و تعداد $10 \times 5 \times 50 = 2500$ عملیات ضرب برای ضرب A_3 در حاصلضرب $A_1 A_2$ باید انجام شود. بنابراین نیاز به انجام 7500 عملیات ضرب است.
- حال فرض کنید پرانتز گذاری به صورت $(A_1 (A_2 A_3))$ باشد. در اینصورت نیاز به انجام $100 \times 5 \times 50 = 25000$ عملیات ضرب برای ضرب $A_2 A_3$ و نیاز به انجام $10 \times 100 \times 50 = 50000$ عملیات ضرب برای ضرب A_1 در حاصلضرب $A_2 A_3$ است، بنابراین در مجموع نیاز به انجام 75000 عملیات ضرب است. بنابراین با استفاده از پرانتز گذاری اول، عملیات ضرب 10^6 برابر سریع‌تر انجام می‌شود.

ضرب زنجیره‌ای ماتریس‌ها

- مسئله ضرب زنجیره‌ای ماتریس‌ها را به صورت زیر بیان می‌کنیم :
زنجیره n ماتریس $\langle A_1, A_2, \dots, A_n \rangle$ را در نظر بگیرید، به طوری که به ازای $i = 1, 2, \dots, n$ ، ابعاد ماتریس A_i برابر است با $p_{i-1} \times p_i$. ضرب $A_1 \cdot A_2 \cdot \dots \cdot A_n$ را طوری پرانتزگذاری کنید که تعداد ضرب‌ها در عملیات ضرب این زنجیره ماتریسی حداقل باشد. ابعاد ورودی مسئله به صورت $\langle p_0, p_1, p_2, \dots, p_n \rangle$ داده شده‌اند.
- در مسئله ضرب زنجیره‌ای ماتریس‌ها نمی‌خواهیم حاصلضرب ماتریس‌ها را به دست آوریم بلکه تنها می‌خواهیم ترتیب ضرب را به گونه‌ای به دست آوریم که هزینه ضرب به حداقل برسد. معمولاً زمانی که صرف پیدا کردن پرانتزگذاری بهینه می‌شود ارزش هزینه کردن دارد، چرا که ممکن است ضرب ماتریس‌ها به صورت ترتیبی هزینه گزافی به کاربر تحمیل کند.

ضرب زنجیره‌ای ماتریس‌ها

- قبل از اینکه این مسئله را حل کنیم، بررسی می‌کنیم چند پرانتز گذاری متفاوت وجود دارد. در واقع یک الگوریتم ساده برای حل این مسئله این است که هزینه همه پرانتز گذاری‌ها را با یکدیگر مقایسه کنیم ولی از آنجایی که تعداد پرانتز گذاری‌ها بسیار زیاد است، بررسی همه حالات مقدور نیست.
- فرض کنید تعداد کل حالات برای پرانتز گذاری n ماتریس برابر باشد با $P(n)$. وقتی $n = 1$ تنها یک ماتریس در زنجیره وجود دارد و بنابراین تنها یک حالت برای پرانتز گذاری وجود دارد. وقتی $n \geq 2$ باشد، درواقع عبارت می‌تواند به دو قسمت شکسته شود به طوری که هر قسمت به طور جداگانه پرانتز گذاری شود. تعداد کل حالت‌های پرانتز گذاری برابر است با ضرب تعداد حالات پرانتز گذاری قسمت اول ضرب در تعداد حالت‌های پرانتز گذاری قسمت دوم.

ضرب زنجیره‌ای ماتریس‌ها

- این زنجیره می‌تواند به شکل‌های متعددی به دو قسمت تقسیم شود که با احتساب همه حالت‌ها عبارت زیر را برای تعداد کل حالت‌های پرانتز گذاری به دست می‌آوریم.

$$P(n) = \begin{cases} 1 & \text{اگر } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{اگر } n \geq 2 \end{cases}$$

- با حل این رابطه بازگشتی به دست می‌آید $P(n) = \Omega(2^n)$. در واقع $P(n)$ دنباله اعداد کاتالان¹ $(1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \dots)$ را می‌سازد که رشد آن نمایی است و بنابراین به ازای n های بسیار بزرگ، بررسی کردن همه حالت‌ها در عمل غیرممکن است.

¹ catalan numbers

ضرب زنجیره‌ای ماتریس‌ها

- حال از روش برنامه‌ریزی پویا برای بهینه‌سازی پرانتز گذاری زنجیرهٔ ماتریسی استفاده می‌کنیم. یک الگوریتم به روش برنامه‌ریزی پویا برای یک مسئلهٔ بهینه‌سازی از چهار مرحله تشکیل شده است :
- ۱- تشخیص ساختار جواب بهینه بر اساس جواب زیرمسئله‌ها
- ۲- تعریف کردن مقدار جواب بهینه به طور بازگشتی
- ۳- محاسبه کردن مقدار جواب بهینه
- ۴- ساختن جواب بهینه توسط اطلاعات محاسبه شده

ضرب زنجیره‌ای ماتریس‌ها

- برای حل یک مسئله بهینه‌سازی توسط برنامه‌ریزی پویا باید مسئله دارای زیرساختار بهینه¹ باشد.
- یک مسئله دارای زیرساختار بهینه است اگر جواب بهینه برای یک مسئله، شامل جواب‌های زیرمسئله‌ها باشد. به عبارت دیگر اگر جواب یک مسئله بهینه‌سازی را به دست آوریم، باید بتوانیم از جواب آن برای زیرمسئله‌ها نیز استفاده کنیم.
- برای مثال مسئله کوتاهترین مسیر در گراف دارای زیرساختار بهینه است. اگر کوتاهترین مسیر از x به y را به دست آوریم به طوری که مسیر از z عبور کند، کوتاهترین مسیر از x به z و همچنین کوتاهترین مسیر از z به y نیز در جواب مسئله به دست آمده است.
- اما مسئله بلندترین مسیر دارای زیرساختار بهینه نیست. اگر بلندترین مسیر از x به y را به دست آوریم به طوری که مسیر از z عبور کند، نمی‌توانیم بگوییم بلندترین مسیر از z به y نیز در جواب مسئله است، زیرا ممکن است بلندترین مسیر از z به y از x عبور کند.

¹ optimal substructure

ضرب زنجیره‌ای ماتریس‌ها

- (گام ۱) تشخیص ساختار جواب بهینه بر اساس جواب زیرمسئله‌ها :
- اولین مرحله در برنامه‌ریزی پویا تشخیص دادن ساختاری از مسئله است که در زیر مسئله‌ها نیز تکرار می‌شود. به عبارت دیگر اگر مسئله را برای یک زیر مسئله حل کنیم، باید بتوانیم با استفاده از اطلاعات زیر مسئله، مسئله را حل کنیم.
- فرض کنید به ازای $i \leq j$ ماتریس $A_{i:j}$ از ضرب ماتریس‌های $A_i A_{i+1} \dots A_j$ به دست بیاید. اگر $i = j$ باشد تنها یک پرانتز گذاری وجود دارد، اما اگر $i < j$ باشد آنگاه برای پرانتز گذاری این عبارت می‌توانیم آن را به دو قسمت $A_{i:k}$ و $A_{k+1:j}$ تقسیم کنیم به طوری که $i \leq k < j$. با ضرب این دو ماتریس در یکدیگر، حاصل $A_{i:j}$ را به دست می‌آوریم. هزینه پرانتز گذاری $A_{i:j}$ برابر است با هزینه پرانتز گذاری $A_{i:k}$ به علاوه هزینه پرانتز گذاری $A_{k+1:j}$ به علاوه هزینه ضرب دو قسمت در یکدیگر.

ضرب زنجیره‌ای ماتریس‌ها

- مسئله ضرب زنجیره‌ای ماتریس‌ها دارای زیرساختار بهینه است. به عبارت دیگر اگر یک پرانتزگذاری برای $A_{i:j}$ پیدا کنیم به طوری که به دو قسمت $A_{i:k}$ و $A_{k+1:j}$ تقسیم شود، پرانتزگذاری $A_{i:k}$ نیز بهینه است (به همین ترتیب پرانتزگذاری $A_{k+1:j}$ نیز بهینه است).
- اثبات: فرض کنیم پرانتزگذاری $A_{i:j}$ بهینه باشد و پرانتزگذاری $A_{i:k}$ بهینه نباشد. در این صورت می‌توانیم یک پرانتزگذاری بهینه برای $A_{i:k}$ پیدا کنیم و آن را در $A_{i:j}$ استفاده کنیم و یک پرانتزگذاری با هزینه کمتر برای $A_{i:j}$ به دست آوریم که با فرض اولیه در تناقض است.

ضرب زنجیره‌ای ماتریس‌ها

- به طور خلاصه، اگر پرانتزگذاری بهینه برای $A_{i:j}$ را پیدا کنیم، این پرانتزگذاری الزاما از دو پرانتزگذاری $A_{i:k}$ و $A_{k+1:j}$ تشکیل شده است و الزاما پرانتزگذاری‌های $A_{i:k}$ و $A_{k+1:j}$ نیز بهینه هستند.
- بدین دلیل می‌توانیم از برنامه‌ریزی پویا استفاده کنیم، زیرا می‌توانیم هزینه‌های پرانتزگذاری‌های $A_{i:k}$ و $A_{k+1:j}$ را از قبل ذخیره کنیم، و از این هزینه‌ها برای محاسبه پرانتزگذاری $A_{i:j}$ استفاده کنیم.

ضرب زنجیره‌ای ماتریس‌ها

- بنابراین باید مقدار k را پیدا کنیم به طوری که هزینه پُرانتز گذاری $A_{i:k}$ به علاوه هزینه پُرانتز گذاری $A_{k+1:j}$ به علاوه هزینه ضرب $A_{i:k}$ در $A_{k+1:j}$ بهینه باشد. آنگاه پُرانتز گذاری $A_{i:j}$ نیز بهینه خواهد بود.
- پس برای حل مسئله یافتن هزینه پُرانتز گذاری بهینه برای $A_{i:j}$ باید به ازای همه k ها هزینه پُرانتز گذاری بهینه برای $A_{i:k}$ و $A_{k+1:j}$ را محاسبه و با هزینه ضرب $A_{i:k}$ در $A_{k+1:j}$ جمع کنیم. آنگاه از این میان k را به گونه‌ای انتخاب کنیم که هزینه پُرانتز گذاری بهینه باشد (تعداد ضرب‌های پُرانتز گذاری $A_{i:j}$ کمترین مقدار ممکن باشد).

ضرب زنجیره‌ای ماتریس‌ها

- (گام ۲) تعریف کردن مقدار جواب بهینه به طور بازگشتی :
- فرض کنید $m[i, j]$ حداقل تعداد ضرب‌های مورد نیاز برای محاسبه $A_{i:j}$ باشد. حداقل تعداد ضرب‌های مورد نیاز برای کل n ماتریس یعنی $A_{1:n}$ برابر است با $m[1, n]$.
- می‌خواهیم یک عبارت بازگشتی برای مقدار $m[i, j]$ محاسبه کنیم.
- اگر $i = j$ باشد، هزینه‌ای وجود ندارد، بنابراین $m[i, j] = 0$.
- اگر $i < j$ باشد، از ساختار جواب بهینه برای زیر مسئله‌ها استفاده می‌کنیم. فرض کنید یک پرانتز گذاری بهینه حاصل ضرب $A_{i:j}$ را به دو قسمت $A_{i:k}$ و $A_{k+1:j}$ تقسیم می‌کند به طوری که $i \leq k < j$. بنابراین $m[i, j]$ برابر است با هزینه $m[i, k]$ برای محاسبه $A_{i:k}$ به علاوه هزینه $m[k+1, j]$ برای محاسبه $A_{k+1:j}$ به علاوه هزینه ضرب دو قسمت در یکدیگر. حاصل ضرب $A_{i:k}A_{k+1:j}$ به تعداد $p_{i-1}p_kp_j$ عملیات ضرب نیاز دارد. بنابراین داریم :

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$$

ضرب زنجیره‌ای ماتریس‌ها

- در رابطه قبل فرض کردیم مقدار k را می‌دانیم، اما از آنجایی که مقدار k ناشناخته است باید همه مقادیر k به ازای $k = i, i + 1, \dots, j - 1$ امتحان کنیم تا مقدار بهینه $m[i, j]$ را به دست آوریم. بنابراین رابطه بازگشتی را در حالت کلی به صورت زیر می‌نویسیم.

$$m[i, j] = \begin{cases} 0 & \text{اگر } i = j \\ \min\{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j : i \leq k < j\} & \text{اگر } i < j \end{cases}$$

ضرب زنجیره‌ای ماتریس‌ها

- (گام ۳) محاسبه کردن مقدار جواب بهینه:

- حال می‌توانیم برنامه‌ای بنویسیم که به صورت بازگشتی رابطه بازگشتی به دست آمده را محاسبه کند تا حداقل مقدار $m[1, n]$ را به دست آوریم. این الگوریتم بازگشتی برای محاسبه، زمانی از مرتبه نمایی نیاز دارد، پس از این الگوریتم نیز در عمل برای n های بسیار بزرگ نمی‌توانیم استفاده کنیم.
- مشکل الگوریتم بازگشتی این است که برخی از زیر مسئله‌ها یعنی برخی از $m[i, j]$ ها ممکن است چندبار محاسبه شوند. برای مثال برای دو پرانتزگذاری $(A_{p:q})(A_{q+1})$ و $(A_{p-1})(A_{p:q})$ دو بار باید هزینه پرانتزگذاری بهینه $A_{p:q}$ محاسبه شود.
- اما تعداد کل زیر مسئله‌ها به ازای $1 \leq i \leq j \leq n$ برابر است با $\Theta(n^2)$.

ضرب زنجیره‌ای ماتریس‌ها

- به جای حل رابطه بازگشتی با استفاده از یک الگوریتم بازگشتی، آن را توسط جدولی حل می‌کنیم که مقادیر $m[i, j]$ را از پایین به بالا محاسبه کند، بدین معنی که از $m[1, 1]$ شروع می‌کنیم و به ترتیب زیر مسئله‌های بزرگ‌تر را با استفاده از زیر مسئله‌های کوچکتر حل می‌کنیم.
- به این روش حل مسئله روش برنامه‌ریزی پویا گفته می‌شود. در برنامه‌ریزی پویا مسئله به زیرمسئله‌ها شکسته شده، و حل مسئله با شروع از کوچکترین زیر مسئله‌ها آغاز می‌شود تا جواب مسئله اصلی با استفاده از زیر مسئله‌های کوچکتر محاسبه می‌شود.

ضرب زنجیره‌ای ماتریس‌ها

- الگوریتم زیر، مسئله بهینه‌سازی ضرب ماتریسی را به روش برنامه‌ریزی پویا حل می‌کند.

Algorithm Matrix Chain

```

function MATRIX-CHAIN-ORDER(p , n)
1: let m[1:n , 1:n] and s[1:n-1 , 2:n] be new tables
2: for i = 1 to n do ▷ chain length 1
3:   m[i,i] = 0
4: for t = 2 to n do ▷ t is the chain length
5:   for i = 1 to n - t + 1 do ▷ chain begins at Ai
6:     j = i + t - 1 ▷ chain ends at Aj
7:     m[i,j] = ∞
8:     for k = i to j - 1 do ▷ try A[i:k] A[k+1 : j]
9:       q = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j]
10:      if q < m[i,j] then
11:        m[i,j] = q ▷ remember this cost
12:        s[i,j] = k ▷ remember this index
13: return m and s

```

ضرب زنجیره‌ای ماتریس‌ها

- زمان مورد نیاز برای حل این مسئله $O(n^3)$ و حافظه مورد نیاز برای حل آن $\Theta(n^2)$ است، زیرا نیاز به نگهداری جدول برای محاسبه زیر مسئله‌ها می‌باشد.
- با استفاده از برنامه‌ریزی پویا، زمان حل یک مسئله را از زمان نمایی به زمان چند جمله‌ای درجه سوم کاهش دادیم.

ضرب زنجیره‌ای ماتریس‌ها

- (گام ۴) ساختن جواب بهینه توسط اطلاعات محاسبه شده :
- گرچه در گام قبل مقدار بهینه برای تعداد ضرب‌ها در یک زنجیرهٔ ماتریسی را محاسبه کردیم، اما روش پرانتزگذاری ماتریس‌ها را به دست نیاوردیم.
- جدول $s[1 : n - 1, 2 : n]$ که در الگوریتم قبل محاسبه کردیم اطلاعات مورد نیاز برای جواب بهینه را نگهداری می‌کند. هر عنصر $s[i, j]$ مقدار k را ذخیره می‌کند، به طوری که $A_{i:j}$ به دو قسمت $A_{i:k}$ و $A_{k+1:j}$ برای ضرب بهینه تقسیم می‌شود.

- الگوریتم زیر پرانتز گذاری را برای مسئله ضرب زنجیره ماتریس‌ها انجام می‌دهد.

Algorithm Print Optimal Parentheses

```
function PRINT-OPTIMAL-PARENS(s, i, j)
1: if i == j then
2:   print "A"i
3: else
4:   print "("
5:   Print-Optimal-Parens (s, i, s[i, j])
6:   Print-Optimal-Parens (s, s[i, j]+1, j)
7:   print ")"
```

ضرب زنجیره‌ای ماتریس‌ها

- برای حل یک مسئله توسط روش برنامه‌ریزی پویا، مسئله باید دو ویژگی داشته باشد.
- ویژگی اول این است که مسئله را باید بتوان با استفاده از جواب زیر مسئله‌های آن به دست آورد. درواقع باید بتوان برای مسئله زیر مسئله‌هایی پیدا کرد که ساختار آنها شبیه مسئله اصلی است. به عبارت دیگر مسئله باید دارای زیرساختار بهینه باشد.
- ویژگی دوم این است که اگر بخواهیم مسئله را توسط الگوریتم بازگشتی حل کنیم باید زیر مسئله‌ها همپوشانی داشته باشند. بدین ترتیب جدول برنامه‌ریزی پویا راه‌حلی برای جلوگیری از محاسبات تکراری در این همپوشانی‌ها خواهد بود.

طولانی‌ترین زیر رشته مشترک

- برخی مواقع زیست شناسان نیاز دارند دو یا چند ارگانیسم مختلف را با یکدیگر مقایسه کنند. برای این کار دی‌ان‌ای این ارگانیسم‌ها باید مقایسه شوند. یک رشته دی‌ان‌ای شامل رشته‌ای از مولکول‌ها به نام مولکول‌های پایه است که می‌توانند آدنین¹، سیتوزین²، گرانین³، یا تیمین⁴ باشند. هریک از این مولکول‌های پایه با یک حرف نشان داده می‌شوند، بنابراین یک رشته دی‌ان‌ای، یک رشته بر روی الفبای $\{A, C, G, T\}$ است. برای مثال $S_1 = ACCGGTC$ یک رشته دی‌ان‌ای است.
- یکی از دلایلی که نیاز داریم دو رشته دی‌ان‌ای را با یکدیگر مقایسه کنیم، برای این است که متوجه شویم دو ارگانیسم چقدر به یکدیگر شباهت دارند.

¹ adenine

² cytosine

³ granine

⁴ thymine

طولانی‌ترین زیر رشته مشترک

- روش‌های مختلفی برای سنجش شباهت دو رشتهٔ دی‌ان‌ای وجود دارند.
- یک روش برای سنجش شباهت این است که زیر رشته‌های مشترک بین دو رشته را پیدا کنیم. هرچقدر این زیر رشته‌های مشترک طول بیشتری داشته باشند، دو رشته به یکدیگر شبیه‌ترند.
- در این روش برای مقایسه دو رشته باید زیر رشته‌های مشترک محاسبه شوند و طولانی‌ترین آنها پیدا شود. به این مسئله، مسئلهٔ پیدا کردن طولانی‌ترین زیر رشته مشترک¹ گفته می‌شود.

¹ longest common subsequence

- یک زیر دنباله از یک دنباله، دنباله‌ای است که از حذف صفر یا بیشتر عنصر از دنباله اصلی به دست بیاید.
- به طور رسمی، به ازای دنباله $X = \langle x_1, x_2, \dots, x_m \rangle$ ، دنباله $Z = \langle z_1, z_2, \dots, z_k \rangle$ را زیر دنباله¹ X می‌نامیم اگر دنباله صعودی $\langle i_1, i_2, \dots, i_k \rangle$ از اندیس‌های X وجود داشته باشند، به طوری که به ازای $j = 1, 2, \dots, k$ داشته باشیم $x_{i_j} = z_j$.
- برای مثال دنباله $Z = \langle B, C, D, B \rangle$ یک زیردنباله از دنباله $X = \langle A, B, C, B, D, A, B \rangle$ است با اندیس‌های $\langle 2, 3, 5, 7 \rangle$.

¹ subsequence

طولانی‌ترین زیر رشته مشترک

- به ازای دو دنباله X و Y ، می‌گوییم دنباله Z یک زیردنباله مشترک X^1 و Y است اگر Z زیر دنباله‌ای از X و همچنین Y باشد.
- برای مثال اگر $X = \langle A, B, C, B, D, A, B \rangle$ و $Y = \langle B, D, C, A, B, A \rangle$ باشند آنگاه $\langle B, C, A \rangle$ یک زیردنباله مشترک X و Y است.
- دنباله $\langle B, C, A \rangle$ با طول ۳ طولانی‌ترین زیر دنباله مشترک X و Y نیست، چرا که دنباله $\langle B, C, B, A \rangle$ با طول ۴ وجود دارد که زیر دنباله مشترک X و Y است. این زیردنباله، طولانی‌ترین زیر دنباله مشترک X^2 و Y است، چرا که زیردنباله مشترک بلندتری وجود ندارد.

¹ common subsequence

² longest common subsequence

طولانی‌ترین زیر رشته مشترک

- می‌توانیم مسئله طولانی‌ترین زیردنبالهٔ مشترک را با استفاده از یک روش جستجوی کامل¹ به دست آوریم، بدین معنی که همهٔ زیردنباله‌های مشترک دو رشته را به دست آوریم و مقایسه کنیم. از آنجایی که رشته X با طول m تعداد 2^m زیردنباله دارد، بنابراین این روش برای رشته‌های طولانی غیر قابل استفاده است.
- می‌خواهیم این مسئله را به روش برنامه‌ریزی پویا حل کنیم.

¹ exhaustive search (brute-force search)

طولانی‌ترین زیر رشته مشترک

- گام اول : مشخص کردن ساختار جواب مسئله بر اساس زیرمسئله‌ها
- برای تعریف مسئله طولانی‌ترین زیررشته مشترک با استفاده از زیر مسئله‌ها، ابتدا مفهوم پیشوند¹ یک دنباله را تعریف می‌کنیم.
- به ازای دنباله $X = \langle x_1, x_2, \dots, x_m \rangle$ ، i امین پیشوند X برابر است با $X_i = \langle x_1, x_2, \dots, x_i \rangle$ به ازای $i = 0, 1, \dots, m$.
- برای مثال اگر $X = \langle A, B, C, B, D, A, B \rangle$ باشد، آنگاه $X_4 = \langle A, B, C, B \rangle$ و X_0 دنباله تهی است.

¹ prefix

طولانی‌ترین زیر رشته مشترک

- فرض کنید $X = \langle x_1, x_2, \dots, x_m \rangle$ و $Y = \langle y_1, y_2, \dots, y_n \rangle$ دو دنباله باشند و $Z = \langle z_1, z_2, \dots, z_k \rangle$ طولانی‌ترین زیررشته مشترک X و Y باشد. خواهیم داشت :

۱- اگر $x_m = y_n$ آنگاه $z_k = x_m = y_n$ و Z_{k-1} طولانی‌ترین زیر رشته مشترک X_{m-1} و Y_{n-1} است.

۲- اگر $x_m \neq y_n$ و $z_k \neq x_m$ باشد، آنگاه Z طولانی‌ترین زیر رشته مشترک X_{m-1} و Y است.

۳- اگر $x_m \neq y_n$ و $z_k \neq y_n$ باشد، آنگاه Z طولانی‌ترین زیر رشته مشترک X و Y_{n-1} است.

طولانی‌ترین زیر رشته مشترک

- گزاره‌های ۱، ۲، ۳ در قضیه قبل را به ترتیب اثبات می‌کنیم.

۱- اگر $x_m = y_n$ آنگاه $z_k = x_m = y_n$ و Z_{k-1} طولانی‌ترین زیر رشته مشترک X_{m-1} و Y_{n-1} است.

اثبات: اگر $x_m = y_n$ باشد، الزاما باید داشته باشیم $z_k = x_m$. این گزاره را با برهان خلف ثابت می‌کنیم. فرض کنید $z_k \neq x_m$ ، آنگاه می‌توانیم x_m که برابر با y_n است را به Z بیافزاییم و زیررشته مشترکی پیدا کنیم که طول آن $k + 1$ است. از آنجایی که در صورت مسئله گفته شده Z طولانی‌ترین زیر رشته مشترک با طول k است، پس به تناقض می‌رسیم. پس فرض اولیه نادرست است و الزاما باید داشته باشیم $z_k = x_m = y_n$. حال باید ثابت کنیم Z_{k-1} طولانی‌ترین زیر رشته مشترک X_{m-1} و Y_{n-1} نیز هست. این گزاره را با برهان خلف ثابت می‌کنیم. فرض یک زیر رشته مشترک W برای X_{m-1} و Y_{n-1} وجود دارد که طول آن از $k - 1$ بیشتر است. در اینصورت با اضافه کردن $x_m = y_n$ به W زیر رشته‌ای ساخته می‌شود که طول آن از k بیشتر است. اما در اینجا به تناقض می‌رسیم چون فرض کردیم طول بلندترین زیر رشته مشترک k است.

طولانی‌ترین زیر رشته مشترک

۲- اگر $x_m \neq y_n$ و $z_k \neq x_m$ باشد، آنگاه Z طولانی‌ترین زیر رشته مشترک X_{m-1} و Y است.

اثبات: اگر $z_k \neq x_m$ باشد، آنگاه Z یک زیر رشته مشترک برای X_{m-1} و Y است. اگر یک زیر رشته مشترک دیگر به نام W با طول بیشتر از k برای X_{m-1} و Y وجود داشت، آنگاه W می‌توانست یک زیر رشته مشترک برای X و Y نیز باشد که این متناقض است با فرض اینکه Z بلندترین زیر رشته مشترک X و Y با طول k است.

۳- اگر $x_m \neq y_n$ و $z_k \neq y_n$ باشد، آنگاه Z طولانی‌ترین زیر رشته مشترک X و Y_{n-1} است.

اثبات: شبیه و مقارن حالت ۲ است.

طولانی‌ترین زیر رشته مشترک

- بنابراین توانستیم مسئله طولانی‌ترین زیر رشته مشترک را بر اساس زیر مسئله‌های بهینه آن تعریف کنیم. جواب زیر مسئله‌های در همهٔ حالت‌های بررسی شده در جواب مسئله وجود دارد.
- پس این مسئله دارای زیرساختار بهینه است.

طولانی‌ترین زیر رشته مشترک

- گام دوم : تعریف کردن مقدار جواب به صورت بازگشتی
- فرض کنید $c[i, j]$ طول بلندترین زیررشته مشترک X_i و Y_j باشد.
- این مسئله بهینه‌سازی را می‌توانیم بر اساس زیر ساختارهای بهینه به صورت زیر تعریف کنیم.

$$c[i, j] = \begin{cases} 0 & \text{اگر } i = 0 \text{ یا } j = 0 \\ c[i - 1, j - 1] + 1 & \text{اگر } x_i = y_j \text{ و } i, j > 0 \\ \max\{c[i, j - 1], c[i - 1, j]\} & \text{اگر } x_i \neq y_j \text{ و } i, j > 0 \end{cases}$$

- دقت کنید که اگر الگوریتم بازگشتی برای حل این مسئله استفاده شود زیر مسئله‌ها به طور تکراری محاسبه می‌شوند. پس می‌توانیم در این جا از برنامه‌ریزی پویا استفاده کنیم.

طولانی‌ترین زیر رشته مشترک

- گام سوم : محاسبه طول طولانی‌ترین زیر دنباله مشترک
- از آنجایی که برای دو دنباله $X = \langle x_1, x_2, \dots, x_m \rangle$ و $Y = \langle y_1, y_2, \dots, y_n \rangle$ مقادیر جدول $c[0 : m, 0 : n]$ باید محاسبه شوند و هر خانه از جدول در زمان ثابت $\Theta(1)$ محاسبه می‌شود، بنابراین زمان اجرای الگوریتم برنامه‌ریزی پویا برای این مسئله برابر است با $\Theta(mn)$. طول زیر رشته مشترک برابر است با مقدار محاسبه شده برای $c[m, n]$.

- الگوریتم طولانی‌ترین زیررشته مشترک به صورت زیر نوشته شده است.

Algorithm Longest Common Subsequence Length

```
function LCS-LENGTH(X,Y, m, n)
1: let b[1:m, 1:n] and c[0:m, 0:n] be new tables
2: for i = 1 to m do
3:   c[i,0] = 0
4: for j = 0 to n do
5:   c[0,j] = 0
```

Algorithm Longest Common Subsequence Length

```

function LCS-LENGTH(X,Y, m, n)
6: for i = 1 to m do ▷ compute table entries in row-major order
7:   for j = 1 to n do
8:     if X[i] == Y[j] then
9:       c[i, j] = c[i-1, j-1] + 1
10:      b[i, j] = "↖"
11:    else if c[i-1, j] ≥ c[i, j-1] then
12:      c[i, j] = c[i-1, j]
13:      b[i, j] = "↑"
14:    else
15:      c[i, j] = c[i, j-1]
16:      b[i, j] = "←"
17: return c and b

```

طولانی‌ترین زیر رشته مشترک

- گام چهارم : ساختن بلندترین زیر دنباله مشترک
- با استفاده از جدول b که توسط الگوریتم قبل ساخته شده می‌توانیم زیر دنباله مشترک X و Y را بسازیم، بدین ترتیب که با $b[m, n]$ شروع می‌کنیم و جهت نشانه‌ها را دنبال می‌کنیم. علامت \searrow در جدول b نشان می‌دهد که $x_i = y_j$ در طولانی‌ترین زیر رشته مشترک است.

طولانی‌ترین زیر رشته مشترک

- برای مثال به ازای دو دنباله $X = \langle A, B, C, B, D, A, B \rangle$ و $Y = \langle B, D, C, A, B, A \rangle$ جدول زیر به دست می‌آید.

		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A	0	↑	0	0	0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2	
3	C	0	↑1	↑1	↖2	←2	↑2	↑2	
4	B	0	↖1	↑1	↑2	↑2	↖3	←3	
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3	
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4	
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4	

Algorithm Print Longest Common Subsequence

– function PRINT-LCS(b, X, i, j)

1: if i == 0 or j == 0 then

2: return ▷ the longest common subsequence has length 0

3: if b[i, j] == "↖" then

4: Print-LCS(b, X, i-1, j-1)

5: print X[i] ▷ same as Y[j]

6: else if b[i, j] = "↑" then

7: Print-LCS(b, X, i-1, j)

8: else

9: Print-LCS(b, X, i, j-1)

طولانی‌ترین زیر رشته مشترک

- پس از طراحی یک الگوریتم معمولاً به دنبال روش‌هایی برای بهبود در زمان اجرا و میزان حافظه می‌گردیم.
- در الگوریتم طولانی‌ترین زیر دنباله مشترک به طور مثال می‌توانیم جدول b را حذف کنیم و اطلاعات لازم برای ساختن بلندترین زیر دنباله مشترک را از جدول c به دست آوریم.
- هریک از درایه‌های $c[i, j]$ از طریق یکی از سه درایه $c[i - 1, j]$ ، $c[i, j - 1]$ ، $c[i - 1, j - 1]$ محاسبه شده است که در زمان ثابت می‌توانیم بدون جدول b به دست آوریم درایه $c[i, j]$ چگونه محاسبه شده است.
- بنابراین طولانی‌ترین زیردنباله مشترک را می‌توانیم همچنان در زمان $\Theta(m + n)$ بسازیم و جدول b را حذف کرده و از حافظه مورد نیاز به میزان mn بکاهیم.

درخت جستجوی دودویی بهینه

- فرض کنید می‌خواهیم برنامه‌ای طراحی کنیم که متون انگلیسی را به فارسی ترجمه کند. به ازای هر کلمه انگلیسی در یک متن باید با استفاده از یک فرهنگ لغت، معادل فارسی آن را بیابیم. برای یک جستجوی بهینه می‌توانیم یک درخت جستجوی دودویی با n رأس بسازیم که هر رأس آن یک کلمه انگلیسی و معادل فارسی آن را شامل شود.
- اگر از یک درخت جستجوی دودویی متوازن¹ استفاده کنیم، می‌توانیم جستجوی هر کلمه را در یک درخت با n کلمه در زمان $O(\lg n)$ انجام دهیم.

¹ balanced binary search tree

- اما کلمات مختلف تعداد تکرارهای مختلف دارند. برای مثال کلمات a یا the در انگلیسی بسیار پر تکرارند و بهتر است این کلمات در درخت جستجو به ریشه نزدیک‌تر باشند و برخی از اسامی خاص بسیار کم تکرارند و بهتر است که فاصله آنها از ریشه بیشتر باشد.
- با استفاده از درخت جستجوی دودویی بهینه¹ می‌توان کلمات را به گونه‌ای ذخیره و بازیابی کرد که کلمات با احتمال وقوع بیشتر نزدیک‌تر به ریشه قرار بگیرند.

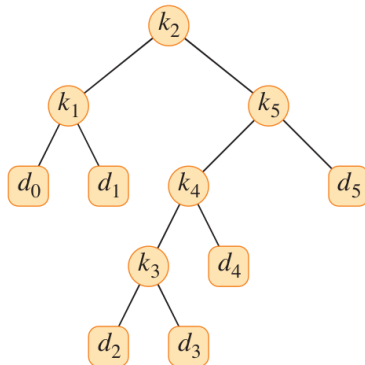
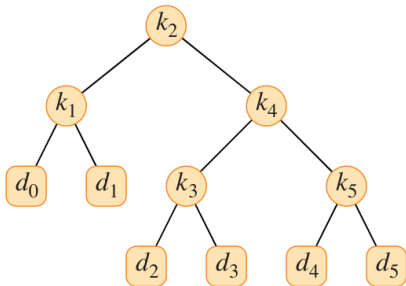
¹ optimal binary search tree

درخت جستجوی دودویی بهینه

- دنباله $K = \langle k_1, k_2, \dots, k_n \rangle$ با n کلید را در نظر بگیرید به طوری که $k_1 < k_2 < \dots < k_n$.
- می‌خواهیم یک درخت جستجوی دودویی بهینه حاوی این کلیدها بسازیم.
- به ازای هر یک از کلیدهای k_i ، یک احتمال وقوع p_i نیز داده شده است.
- از آنجایی که برخی از کلیدها در درخت جستجو وجود ندارد (برای مثال کلماتی در کاربرد ترجمه در انگلیسی وجود دارند که معادل فارسی ندارند)، تعداد $n + 1$ کلید بی‌استفاده $d_0, d_1, d_2, \dots, d_n$ نیز داریم که نماینده این کلیدها هستند. در واقع d_0 نماینده همه کلیدهایی است که از k_1 کوچکترند و d_n نماینده همه کلیدهایی است که از k_n بزرگترند و همچنین به ازای $i = 1, 2, \dots, n - 1$ ، کلید d_i نماینده همه مقادیری است که بین k_i و k_{i+1} قرار دارند. همچنین به ازای هر کلید d_i یک احتمال وقوع q_i داریم.

درخت جستجوی دودویی بهینه

- در شکل زیر دو درخت جستجوی دودویی بهینه را با تعداد ۵ کلید مشاهده می‌کنیم.



درخت جستجوی دودویی بهینه

- هر یک از کلیدهای k_i یک رأس میانی است و هر یک از کلیدهای بی‌استفاده d_i یک برگ در درخت جستجوی بهینه است.
- از آنجایی که هر جستجو یا موفق است (که منجر به پیدا کردن یک کلید k_i می‌شود) و یا ناموفق (که منجر به رسیدن به کلید بی‌استفاده d_i است)، بنابراین داریم :

$$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$$

درخت جستجوی دودویی بهینه

- با اطلاع داشتن از احتمال وقوع هر یک از کلیدها، می‌توانیم هزینه جستجو در یک درخت جستجو را پیدا کنیم.
- فرض کنید هزینه جستجوی یک کلید در درخت به ازای هر بار جستجو برابر با تعداد رئوس بررسی شده برای رسیدن به آن کلید باشد. بنابراین هزینه جستجوی یک کلید در یک جستجو برابر خواهد بود با عمق¹ رأس مربوط به آن کلید به علاوه یک. ریشه در عمق صفر قرار دارد، بنابراین هزینه یافتن کلید مربوط به ریشه در یک جستجو برابر است با یک.
- برای یافتن هزینه جستجوی یک کلید در یک متن، باید هزینه یک بار جستجو را در احتمال وقوع آن کلید ضرب کنیم.
- نهایتاً برای یافتن هزینه جستجوی یک درخت باید هزینه جستجوی همه کلیدها را با هم جمع کنیم.

¹ depth

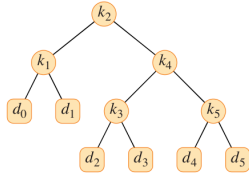
- بنابراین هزینه جستجو در درخت T برابر است با :

$$\begin{aligned} E[\text{search cost in } T] &= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{depth}_T(d_i) \cdot q_i \end{aligned}$$

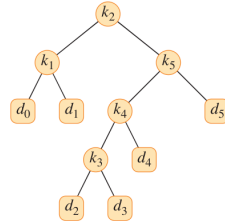
- در اینجا depth_T تابعی است که عمق یک کلید را در درخت T نشان می‌دهد.

درخت جستجوی دودویی بهینه

- در شکل زیر هزینه جستجو برای دو درخت جستجو محاسبه شده است.



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
Total			2.75

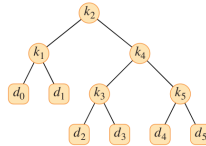
درخت جستجوی دودویی بهینه

- حال به ازای تعدادی کلید به همراه احتمال وقوع آنها، می‌خواهیم یک درخت جستجوی دودویی بیابیم که هزینه جستجو در آن حداقل است.
- به این درخت، درخت جستجوی دودویی بهینه¹ گفته می‌شود.

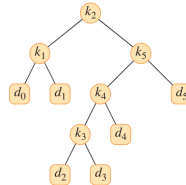
¹ optimal binary search tree

درخت جستجوی دودویی بهینه

- در شکل زیر دو درخت جستجوی دودویی نشان داده شده‌اند. هزینه جستجو در درخت سمت چپ 2.80 و در درخت سمت راست برابر با 2.75 است. درخت سمت راست یک درخت جستجوی بهینه است. در اینجا می‌توانیم ببینیم درخت جستجوی دودویی بهینه، الزاماً درختی نیست که عمق آن کمتر باشد.



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
Total			2.75

درخت جستجوی دودویی بهینه

- همچنین درخت جستجوی دودویی بهینه الزاماً درختی نیست که همه کلیدها با احتمال وقوع بیشتر در ریشه آن باشند. برای مثال کلید k_5 بیشترین احتمال وقوع را دارد، اما ریشه درخت جستجو k_2 است.

- همانند مسئله ضرب زنجیره‌ای ماتریس‌ها، با استفاده از جستجوی کامل برای بررسی همه درخت‌های جستجو نمی‌توانیم در زمان چندجمله‌ای درخت جستجوی دودویی بهینه را به دست آوریم. تعداد همه درخت‌های جستجوی دودویی از مرتبه نمایی است، بنابراین بررسی همه درخت‌های جستجو ممکن نیست.
- می‌خواهیم این مسئله را با استفاده از برنامه‌ریزی پویا حل کنیم.

درخت جستجوی دودویی بهینه

- گام اول : ساختار یک درخت جستجوی دودویی بهینه
- برای بررسی ساختار و مشخص کردن ویژگی‌های یک درخت بهینه، ابتدا ساختار درخت و زیردرخت‌های آن را بررسی می‌کنیم. در واقع باید اثبات کنیم این مسئله دارای زیرساختار بهینه است، بدین معنی که جواب زیرمسئله‌ها را می‌توان از جواب مسئله استخراج کرد.
- اگر یک درخت جستجوی دودویی بهینه T داشته باشیم، زیر درخت T' نیز باید بهینه باشد. اگر یک زیر درخت T'' وجود داشت که هزینه آن کمتر از T' بود، می‌توانستیم T' را با T'' جایگزین کنیم و یک درخت با هزینه کمتر به جای T بیابیم که با فرض بهینه بودن T در تناقض است.
- حال می‌خواهیم مسئله را با استفاده از جواب زیر مسئله‌های بهینه آن حل کنیم.

درخت جستجوی دودویی بهینه

- یک زیردرخت دلخواه را در نظر بگیرید. این زیردرخت شامل کلیدهای k_i, \dots, k_j است که برگ‌های آن را کلیدهای d_{i-1}, \dots, d_j تشکیل می‌دهند، به طوری که $1 \leq i \leq j \leq n$.
- به ازای کلیدهای k_i, \dots, k_j ، یکی از این کلیدها، برای مثال کلید k_r به طوری که $i \leq r \leq j$ ، ریشه زیردرخت بهینه برای این کلیدهاست.
- زیردرخت سمت چپ ریشه k_r شامل کلیدهای k_i, \dots, k_{r-1} و کلیدهای برگ d_{i-1}, \dots, d_{r-1} است و زیردرخت سمت راست شامل کلیدهای k_{r+1}, \dots, k_j و کلیدهای برگ d_r, \dots, d_j است.
- فرض کنید در یک زیردرخت با کلیدهای k_i, \dots, k_j ، کلید k_i را به عنوان ریشه انتخاب کنیم. زیردرخت سمت چپ این زیردرخت یک برگ با کلید d_{i-1} است.
- به طور مشابه، اگر k_j را به عنوان ریشه در نظر بگیریم زیر درخت سمت راست این زیر درخت شامل یک برگ با کلید d_j است.

درخت جستجوی دودویی بهینه

- گام دوم : راه حل بازگشتی
- حال برای تعریف راه حل بهینه به صورت بازگشتی، زیر درختی شامل کلیدهای k_i, \dots, k_z را در نظر بگیرید به طوری که $i \geq 1$ ، $z \leq n$ و $z \geq i - 1$. وقتی $z = i - 1$ باشد، تنها کلید برگ d_{i-1} را خواهیم داشت.
- فرض کنید $e[i, z]$ هزینه جستجوی یک درخت جستجوی بهینه با کلیدهای k_i, \dots, k_z باشد. هدف محاسبه هزینه جستجو برای همه کلیدهاست که برابر با مقدار $e[1, n]$ می باشد.
- اگر $z = i - 1$ باشد، آنگاه مسئله تنها شامل یک کلید d_{i-1} می شود. در این صورت هزینه جستجو برابر است با $e[i, i - 1] = q_{i-1}$.
- وقتی $z \geq i$ باشد، باید ریشه k_r را از بین کلیدهای k_i, \dots, k_z انتخاب کنیم و یک درخت جستجوی بهینه با کلیدهای k_i, \dots, k_{r-1} به عنوان زیردرخت سمت چپ ریشه k_r و یک درخت جستجوی بهینه با کلیدهای k_{r+1}, \dots, k_z به عنوان زیردرخت سمت راست ریشه k_r بسازیم.

درخت جستجوی دودویی بهینه

- وقتی یک زیردرخت بهینه T' به عنوان زیردرخت یک رأس قرار می‌گیرد و درخت T را تشکیل می‌دهد، درواقع عمق هر یک از رأس‌های T' در درخت T یک واحد افزوده می‌شود. در اینصورت هزینه جستجو برای رئوس زیردرخت T' در درخت T به میزان مجموع احتمال رئوس T' افزایش می‌یابد.

درخت جستجوی دودویی بهینه

- برای مثال فرض کنید درخت T' با کلیدهای k_1, k_2, k_3 را تشکیل داده باشیم. هزینه جستجوی این درخت (بدون در نظر گرفتن کلیدهای بی‌استفاده) برابر است با $E[T'] = \sum_{i=1}^3 (\text{depth}_{T'}(k_i) + 1) \cdot p_i$.
- اگر زیردرخت T' در درخت T قرار بگیرد به طوری که ریشه درخت T کلید k_4 و T' زیردرخت سمت چپ در درخت T باشد، آنگاه خواهیم داشت:

$$\begin{aligned} E[T] &= p_4 + \left(\sum_{i=1}^3 (\text{depth}_T(k_i) + 1) \cdot p_i \right) \\ &= p_4 + \left(\sum_{i=1}^3 (\text{depth}_{T'}(k_i) + 2) \cdot p_i \right) \\ &= p_4 + E[T'] + \left(\sum_{i=1}^3 p_i \right) \end{aligned}$$

- برای یک زیردرخت با کلیدهای k_i, \dots, k_j مجموع احتمال‌ها برابر است با :

$$w(i, j) = \sum_{l=i}^j p_l + \sum_{l=i-1}^j q_l$$

- بنابراین اگر k_r ریشه یک زیردرخت بهینه با کلیدهای k_i, \dots, k_j باشد، خواهیم داشت :

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$$

درخت جستجوی دودویی بهینه

- از آنجایی که مجموع احتمال وقوع همهٔ رئوس در یک درخت برابر است با مجموع احتمال‌های وقوع رئوس زیردرخت چپ به علاوهٔ احتمال وقوع ریشه به علاوهٔ احتمال‌های وقوع رئوس زیردرخت راست، بنابراین رابطه زیر برقرار است :

$$w(i, j) = w(i, r - 1) + p_r + w(r + 1, j)$$

- بنابراین می‌توانیم رابطه بازگشتی برای محاسبه هزینهٔ جستجو در درخت بهینه را به صورت زیر بازنویسی کنیم :

$$e[i, j] = e[i, r - 1] + e[r + 1, j] + w(i, j)$$

درخت جستجوی دودویی بهینه

- در اینجا فرض کردیم که می‌دانیم کدام رأس به عنوان رأس ریشه k_r انتخاب می‌شود.
- از آنجایی که هدف این است که ریشه‌ای را انتخاب کنیم که مقدار هزینه جستجو را کاهش دهد، بنابراین رابطه بازگشتی برای محاسبه هزینه جستجو در درخت بهینه را به صورت زیر می‌نویسیم.

$$e[i, j] = \begin{cases} q_{i-1} & \text{اگر } j = i - 1 \\ \min\{e[i, r - 1] + e[r + 1, j] + w(i, j) : i \leq r \leq j\} & \text{اگر } i \leq j \end{cases}$$

- بنابراین رابطه‌ای برای جدول $e[i, j]$ جهت استفاده در یک الگوریتم برنامه‌ریزی پویا به صورت بازگشتی محاسبه کردیم.

درخت جستجوی دودویی بهینه

- جدول $e[i, j]$ تنها میزان هزینه جستجوی بهینه را نگهداری می‌کند.
- به یک جدول دیگر نیاز داریم برای اینکه بتوانیم ساختار درخت را نیز نگهداری کنیم تا در نهایت بتوانیم درخت جستجو بهینه را بازسازی کنیم.
- این اطلاعات را در جدول $root[i, j]$ نگهداری می‌کنیم. درواقع مقدار $root[i, j]$ به ازای $1 \leq i \leq j \leq n$ برابر است با اندیس r برای کلید k_r که ریشه درخت جستجوی بهینه‌ای است که برای کلیدهای k_i, \dots, k_j یافته می‌شود.

درخت جستجوی دودویی بهینه

- گام سوم : محاسبه هزینه جستجو در یک درخت جستجوی دودویی بهینه
- حال با استفاده از روش برنامه‌ریزی پویا می‌توانیم مقادیر $e[i, j]$ را به ترتیب از پایین به بالا محاسبه می‌کنیم. بنابراین کل جدول را به ازای $e[1 : n + 1, 0 : n]$ مقداردهی می‌کنیم.
- اندیس اول از 1 شروع شده و با $n + 1$ خاتمه می‌یابد، زیرا برای داشتن یک زیردرخت شامل تنها کلید d_n نیاز داریم $e[n + 1, n]$ را محاسبه کنیم. اندیس دوم باید از صفر شروع شود، زیرا برای داشتن یک زیردرخت تنها با کلید d_0 ، باید مقدار $e[1, 0]$ را محاسبه کنیم.
- همه مقادیر $e[i, j]$ به ازای $j \geq i - 1$ باید محاسبه شوند. جدول $root[i, j]$ ریشه زیردرخت‌ها را با کلیدهای k_i, \dots, k_j ذخیره می‌کند، به طوری که $1 \leq i \leq j \leq n$.

درخت جستجوی دودویی بهینه

- همچنین می‌توانیم از یک جدول دیگر بهره بگیریم تا محاسبات را سریع‌تر انجام دهیم.
- به جای محاسبه $w(i, j)$ برای هر یک از درایه‌های $e[i, j]$ جدول $w[1 : n + 1, 0 : n]$ را محاسبه می‌کنیم.
در حالت پایه، مقدار $w[i, i - 1] = q_{i-1}$ به ازای $1 \leq i \leq n + 1$
- به ازای $i \geq j$ درایه‌های جدول w را به صورت زیر محاسبه می‌کنیم.
$$w[i, j] = w[i, j - 1] + p_j + q_j$$
- بنابراین می‌توانیم $\Theta(n^2)$ مقدار $w[i, j]$ را هرکدام در زمان $\Theta(1)$ محاسبه کنیم.

- الگوریتم زیر مسئله درخت جستجوی دودویی بهینه را به روش برنامه‌ریزی پویا حل می‌کند.

Algorithm Optimal-BST

```
function OPTIMAL-BST(p, q, n)
1: let e[1:n+1 , 0:n], w[1:n+1 , 0:n], and root[1:n+1 , 1:n] be new tables
2: for i = 1 to n + 1 do ▷ base cases
3:   e[i,i-1] = q[i-1]
4:   w[i,i-1] = q[i-1]
```

Algorithm Optimal-BST

```

-   function OPTIMAL-BST(p, q, n)
5:   for t = 1 to n do
6:     for i = 1 to n - t + 1 do
7:       j = i + t - 1
8:       e[i,j] =  $\infty$ 
9:       w[i,j] = w[i,j-1] + p[j] + q[j]
10:      for r = i to j do          ▷ try all possible roots r
11:        t = e[i,r-1] + e[r+1,j] + w[i,j]
12:        if t < e[i,j] then      ▷ new minimum?
13:          e[i,j] = t
14:          root[i,j] = r
15: return e and root

```

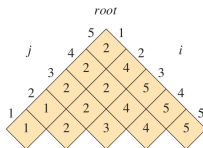
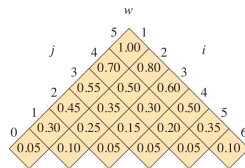
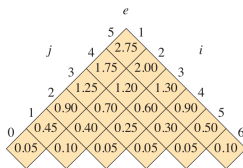
درخت جستجوی دودویی بهینه

- الگوریتم درخت جستجوی دودویی بهینه محاسبات را در زمان $\Theta(n^3)$ انجام می‌دهد و به یک جدول با اندازه $\Theta(n^2)$ نیاز دارد.

درخت جستجوی دودویی بهینه

- در شکل زیر، جدول‌های $e[i, j]$ ، $w[i, j]$ و $root[i, j]$ با استفاده از الگوریتم برنامه‌ریزی پویا برای جستجوی دودویی بهینه برای کلیدهای تعیین شده زیر، محاسبه شده‌اند.

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10



کوله‌پشتی ۱-۰

- یک دزد با یک کوله‌پشتی به یک فروشگاه دستبرد می‌زند. وزنی که کوله‌پشتی او می‌تواند تحمل کند W است. در این مغازه تعداد n کالا وجود دارد. هر کالای $item_i$ دارای وزن w_i و ارزش v_i است. دزد می‌خواهد از میان این کالاها تعدادی را انتخاب کرده در کوله‌پشتی خود قرار دهد به طوری که مجموع وزن کالاهای انتخاب شده از ظرفیت کوله‌پشتی یعنی W بیشتر نباشد و مجموع ارزش کالاهای دزدیده شده حداکثر باشد.
- بنابراین دزد می‌خواهد از مجموعه $S = \{item_1, item_2, \dots, item_n\}$ یک زیر مجموعه A را انتخاب کند به طور $\sum_{item_i \in A} v_i$ بیشترین مقدار ممکن باشد و $\sum_{item_i \in A} w_i \leq W$ باشد.
- تعداد همه حالت‌های ممکن تعداد همه زیر مجموعه‌های S است که برابر است با 2^n جایی که n تعداد کالاهاست.
- در این مسئله دزد یا می‌تواند یک کالا را بردارد یا بگذارد و امکان شکستن کالاهای به دو قسمت وجود ندارد. به همین دلیل به آن مسئله کوله‌پشتی ۱-۰^۱ گفته می‌شود. در مسئله کوله‌پشتی کسری^۲ دزد می‌تواند یک کالا را به دو قسمت تقسیم کرده، یک قسمت را در کوله‌پشتی قرار دهد و قسمت دیگر را در مغازه بگذارد.

^۱ 0-1 knapsack

^۲ fractional knapsack

- در گام اول باید اثبات کنیم این مسئله دارای زیر ساختار بهینه است یا به عبارت دیگر قانون بهینگی^۱ برای آن صادق است.
- فرض کنید A زیر مجموعه بهینه از n کالا باشد. دو حالت وجود دارد: یا A شامل $item_n$ می‌شود یا خیر. اگر A کالای $item_n$ را شامل نشود، A یک زیر مجموعه بهینه برای $n - 1$ کالا نیز هست.
- اگر A کالای $item_n$ را شامل شود، آنگاه مجموع ارزش‌های کالاهای A برابر است با v_n به علاوه بیشترین ارزش ممکن که از $n - 1$ کالا برای یک کوله‌پشتی با ظرفیت $W - w_n$ به دست آمده است. این گزاره‌ها را می‌توانیم با برهان خلف اثبات کنیم.

^۱ principle of optimality

- در گام دوم باید یک رابطه بازگشتی برای محاسبه جواب مسئله براساس جواب زیر مسئله‌ها بنویسیم.
- فرض کنید $P[i][w]$ بیشترین ارزش به دست آمده از i کالای اول است وقتی که ظرفیت کوله‌پشتی w باشد.
- می‌توانیم یک رابطه بازگشتی به صورت زیر برای محاسبه $P[i][w]$ بنویسیم.

$$P[i][w] = \begin{cases} \max(P[i-1][w], v_i + P[i-1][w - w_i]) & \text{اگر } w_i \leq w \\ P[i-1][w] & \text{اگر } w_i > w \end{cases}$$

- در این مسئله به دنبال $P[n][W]$ می‌گردیم.

- می‌توانیم جدولی تشکیل دهیم که هر سطر i در آن نشان دهنده این باشد که فقط از i کالای اول استفاده کرده‌ایم و ستون‌های آن همه وزن‌های ممکن از 0 تا W باشد.
- مقادیر $P[i][0]$ و $P[0][w]$ برابر با صفر هستند.
- این جدول دارای nW خانه است پس محاسبه این جدول در زمان $\Theta(nW)$ امکان‌پذیر است.

- توجه کنید که هیچ رابطه‌ای بین n و W وجود ندارد و این الگوریتم می‌تواند از الگوریتمی که همه حالات را بررسی می‌کند بدتر باشد. برای مثال اگر $W = n!$ باشد الگوریتم برنامه‌ریزی پویا از مرتبه $n!$ است درحالی که بررسی همه حالات در زمان $\Theta(2^n)$ امکان‌پذیر است.
- بنابراین تنها در صورتی از برنامه‌ریزی پویا استفاده می‌کنیم که $nW < 2^n$ باشد.
- پیچیدگی زمانی $\Theta(nW)$ گرچه شبیه به پیچیدگی زمانی چندجمله‌ای است، اما در واقع چندجمله‌ای نیست و مقدار W می‌تواند یک تابع غیرچندجمله‌ای از ورودی مسئله باشد. این پیچیدگی زمانی را پیچیدگی زمانی شبه چندجمله‌ای^۱ می‌نامیم.

^۱ pseudo-polynomial time complexity

سریع‌ترین مسیر در جدول

- جدول A با m سطر و n ستون را در نظر بگیرید به طوری که مقدار هرکدام از درایه‌های جدول یک عدد صحیح است.
- می‌خواهیم مهره‌ای را با شروع از درایه $(1, 1)$ حرکت داده، به درایه (m, n) منتقل کنیم. فرض کنید درایه $(1, 1)$ در شمال غرب و درایه (m, n) در جنوب شرق جدول قرار دارد.
- وقتی مهره وارد درایه (i, j) می‌شود، باید $A[i, j]$ ثانیه در آن درایه صبر کند و پس از آن به حرکت ادامه دهد.
- با فرض اینکه مهره تنها می‌تواند به سمت جنوب یا شرق یا مورب به جنوب شرق حرکت کند، می‌خواهیم کمترین زمان ممکن برای انتقال مهره از درایه $(1, 1)$ به (m, n) را محاسبه کنیم.

سریع‌ترین مسیر در جدول

- در گام اول بررسی می‌کنیم مسئله دارای زیر ساختار بهینه است یا اصل بهینگی در آن برقرار است.
- اگر از درایه $(1, 1)$ مهره را حرکت داده در کمترین زمان ممکن به درایه (i, j) برسیم حتماً از یکی از سه درایه $(i-1, j)$ یا $(i, j-1)$ یا $(i-1, j-1)$ عبور کرده‌ایم.
- در صورتی که از $(i-1, j)$ عبور کرده باشیم الزاماً برای حرکت از درایه $(1, 1)$ به $(i-1, j)$ کمترین زمان ممکن را صرف کرده‌ایم. این گزاره را می‌توانیم با برهان خلف اثبات کنیم.
- به همین ترتیب ممکن است از درایه‌های $(i, j-1)$ یا $(i-1, j-1)$ عبور کرده باشیم که به طور مشابه می‌توانیم اثبات کنیم الزاماً در کمترین زمان ممکن به این درایه‌ها رسیده‌ایم.

سریع‌ترین مسیر در جدول

- در گام دوم رابطه‌ای برای توصیف جواب مسئله براساس جواب زیر مسئله‌ها به صورت زیر می‌نویسیم.

$$T[i, j] = \begin{cases} \min(T[i-1, j], T[i, j-1], T[i-1, j-1]) + A[i, j] & \text{اگر } j > 1, i > 1 \\ T[i-1, 1] + A[i, 1] & \text{اگر } j = 1, i > 1 \\ T[1, j-1] + A[1, j] & \text{اگر } i = 1, j > 1 \\ A[1, 1] & \text{اگر } j = 1, i = 1 \end{cases}$$

سریع ترین مسیر در جدول

- سپس جدول T را در زمان $\Theta(mn)$ تکمیل می کنیم و $T[m, n]$ جواب مسئله است.