به نام خدا

طراحی الگوریتم‌ها

آرش شفیعی

برنامه‌ریزی پویا

برنامه‌ریزی پویا

- برنامه‌ریزی پویا [1] روشی دیگر برای حل مسائل محاسباتی است که توسط آن همانند روش تقسیم و حل جواب یک مسئله با ترکیب 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 یک زیر مسئله یک بار حل می‌شود و در یک جدول ذخیره می‌شود و از جواب آن زیر مسئله برای حل زیر مسئله‌های دیگر با اندازه‌های بزرگتر استفاده می‌شود. در واقع در هر مرحله یک زیر مسئلهٔ بزرگ‌تر با استفاده از جواب یک زیر مسئله کوچک‌تر حل می‌شود و این روند ادامه پیدا می‌کند تا اینکه مسئلهٔ اصلی با استفاده از بزرگترین زیرمسئلهٔ به دست آمده حل می‌شود.
- برنامه‌ریزی پویا در بسیاری از مسائل بهینه سازی [2] کاربرد دارد. چنین مسئله‌هایی معمولا چند جواب دارند که ما به دنبال جوابی می‌گردیم که مقدار بهینه (کوچکترین یا بزرگترین) داشته باشد.

---

[1] dynamic programming
[2] optimization problem

ضرب زنجیره‌ای ماتریس‌ها

- مسئلهٔ ضرب زنجیره‌ای ماتریس‌ها [1] به صورت زیر است. می‌خواهیم دنباله (زنجیره)ای از n ماتریس $\langle A_1, A_2, \cdots, A_n \rangle$ 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 شرکت پذیر [2] ، بدین معنی که پرانتز گذاری 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 حاصلضرب C در اینصورت 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 پرانتزگذاری به صورت $((A_1A_2)A_3)$ باشد، تعداد $10 \times 100 \times 5 = 5000$ عملیات ضرب برای ضرب A_1A_2 و تعداد $10 \times 5 \times 50 = 2500$ عملیات ضرب برای ضرب A_3 در حاصلضرب A_1A_2 باید انجام شود. بنابراین نیاز به انجام 7500 عملیات ضرب است.

- حال فرض کنید پرانتزگذاری به صورت $(A_1(A_2A_3))$ باشد. در اینصورت نیاز به انجام $100 \times 5 \times 50 = 25000$ عملیات ضرب برای ضرب A_2A_3 و نیاز به انجام $10 \times 100 \times 50 = 50000$ عملیات ضرب برای ضرب A_1 در حاصلضرب A_2A_3 است، بنابراین در مجموع نیاز به انجام 75000 عملیات ضرب است. بنابراین با استفاده از پرانتزگذاری اول، عملیات ضرب ۱۰ برابر سریع‌تر انجام می‌شود.

ضرب زنجیره‌ای ماتریس‌ها

- مسئله ضرب زنجیره‌ای ماتریس‌ها را به صورت زیر بیان می‌کنیم :
زنجیرهٔ n ماتریس $\langle A_1, A_2, \cdots, A_n \rangle$ را در نظر بگیرید، به طوری‌که به ازای $i = 1, 2, \cdots, n$ ، ابعاد ماتریس A_i برابر است با $p_{i-1} \times p_i$. ضرب $A_1 \cdot A_2 \cdots A_n$ را طوری پرانتز گذاری کنید که تعداد ضرب‌ها در عملیات ضرب این زنجیرهٔ ماتریسی حداقل باشد. ابعاد ورودی مسئله به صورت $\langle p_0, p_1, p_2, \cdots, p_n \rangle$ 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. معمولا زمانی که صرف پیدا کردن پرانتز گذاری بهینه می‌شود ارزش هزینه کردن دارد، چرا که ممکن است ضرب ماتریس‌ها به صورت ترتیبی هزینهٔ گزافی به کاربر تحمیل کند.

ضرب زنجیره‌ای ماتریس‌ها

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

ضرب زنجیره‌ای ماتریس‌ها

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

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

- با حل این رابطهٔ بازگشتی به دست می‌آید $P(n) = \Omega(2^n)$. در واقع $P(n)$ دنبالهٔ اعداد کاتالان [1] (۱، ۱، ۲، ۵، ۱۴، ۴۲، ۱۳۲، ۴۲۹، ۱۴۳۰، ۴۸۶۲، ۱۶۷۹۶، ۵۸۷۸۶، ...) را می‌سازد که رشد آن نمایی است و بنابراین به ازای n های بسیار بزرگ، بررسی کردن همهٔ 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- مشخص کردن ساختار جواب بهینه

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

- (گام ۱) ساختار پرانتز گذاری بهینه :
- 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 $i \leqslant j$ ماتریس $A_{i:j}$ از ضرب ماتریس‌های $A_i A_{i+1} \cdots A_j$ 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 طوری‌که $i \leqslant k < j$. با ضرب این دو ماتریس در یکدیگر، حاصل $A_{i:j}$ را به دست می‌آوریم. هزینهٔ پرانتز گذاری $A_{i:j}$ برابر است با هزینه پرانتز گذاری $A_{i:k}$ به علاوهٔ هزینهٔ پرانتز گذاری $A_{k+1:j}$ به علاوهٔ هزینهٔ ضرب دو قسمت در یکدیگر.

ضرب زنجیره‌ای ماتریس‌ها

- بنابراین باید مقدار k را پیدا کنیم به طوری‌که هزینهٔ پرانتز گذاری $A_{i:k}$ به علاوهٔ هزینهٔ پرانتزگذاری $A_{k+1:j}$ به علاوهٔ هزینهٔ ضرب $A_{i:k}$ در $A_{k+1:j}$ بهینه باشد. آنگاه پرانتز گذاری $A_{i:j}$ نیز بهینه خواهد بود.

ضرب زنجیره‌ای ماتریس‌ها

- (گام ۲) راه حل 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 $A_{1:n}$ برابراست با $m[1,n]$.
- می‌خواهیم یک عبارت بازگشتی برای محاسبه $m[i,j]$ 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 حاصلضرب $A_{i:j}$ را به دو قسمت $A_{i:k}$ و $A_{k+1:j}$ تقسیم می‌کند به طوری‌که $i \leqslant k < j$. بنابراین $m[i,j]$ برابر است با هزینه $m[i,k]$ برای محاسبه $A_{i:k}$ به علاوه هزینه $m[k+1,j]$ برای محاسبهٔ $A_{k+1:j}$ به علاوه هزینهٔ ضرب دو قسمت در یکدیگر. حاصلضرب $A_{i:k}A_{k+1:j}$ 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 همهٔ مقادیر k به ازای $k = i, i+1, \cdots, j-1$ امتحان کنیم تا مقدار بهینه $m[i,j]$ را به‌دست آوریم. بنابراین رابطهٔ بازگشتی را در حالت کلی به صورت زیر می‌نویسیم.

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

ضرب زنجیره‌ای ماتریس‌ها

- (گام ۳) محاسبهٔ هزینهٔ بهینه:
- حال می‌توانیم برنامه‌ای بنویسیم که به صورت بازگشتی رابطهٔ بازگشتی به دست آمده را محاسبه کند تا حداقل مقدار $m[1, n]$ را به‌دست آورد. 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.
- اما تعداد کل زیر مسئله‌ها به ازای $1 \leqslant i \leqslant j \leqslant n$ برابراست 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 حل 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 زیرمسأله‌ها شکسته شده، و حل مسئله با شروع از کوچکترین زیر مسئله‌ها آغاز می‌شود تا جواب مسئلهٔ اصلی با استفاده از زیر مسئله‌های کوچکتر محاسبه می‌شود.

ضرب زنجیره‌ای ماتریس‌ها

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

ضرب زنجیره‌ای ماتریس‌ها

Algorithm Matrix Chain

```
function MATRIX-CHAIN-ORDER(p , n)
let m[1:n , 1:n] and s[1:n-1 , 2:n] be new tables
for i = 1 to n do  ▷ chain length 1
   m[i,i] = 0
for t = 2 to n do  ▷ l is the chain length
   for i = 1 to n - t + 1 do     ▷ chain begins at Ai
      j = i + t - 1        ▷ chain end at Aj
      m[i,j] = ∞
      for k = i to j - 1 do        ▷ try A[i:k] A[k+1 : j]
         q = m[i,k] + m[k+1,j] + p[i-1]p[k]p[j]
         if q < m[i,j] then
            m[i,j] = q              ▷ remember this cost
            s[i,j] = k              ▷ remember this index
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 زمان حل یک مسئله را از زمان نمایی به زمان چند جمله‌ای درجه سوم کاهش دادیم.

ضرب زنجیره‌ای ماتریس‌ها

- (گام ۴) ساختن راه‌حل بهینه :
- 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.

ضرب زنجیره‌ای ماتریس‌ها

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

Algorithm Print Optimal Parentheses

```
function PRINT-OPTIMAL-PARENS(s, i, j)
if i == j then
    print "A"i
else
    print "("
    Print-Optimal-Parens (s, i, s[i, j])
    Print-Optimal-Parens (s, s[i, j]+1, j)
    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. 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 به یکدیگر مقایسه کنند. برای این کار دی‌ان‌ای این ارگانیسم‌ها باید مقایسه شوند. یک رشتهٔ دی‌ان‌ای شامل رشته‌ای از مولکول‌ها به نام مولکول‌های پایه است که می‌توانند آدنین [1] ، سیتوزین [2] ، گرانین [3] ، یا تیمین [4] 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 رشتهٔ 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

- یک روش برای سنجش شباهت دو رشتهٔ دی‌ان‌ای این است که برای شباهت یک مقدار عددی تعریف کنیم. این مقدار عددی برابر است با تعداد تغییرات مورد نیاز در یک رشته برای به دست آوردن رشتهٔ دیگر.
- یک روش دیگر 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 طولانی‌ترین زیر رشته مشترک [1] گفته می‌شود.

[1] longest common subsequence

طولانی‌ترین زیر رشته مشترک

- 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 دنبالهٔ $X = \langle x_1, x_2, \cdots, x_m \rangle$ ، دنبالهٔ $Z = \langle z_1, z_2, \cdots, z_k \rangle$ را زیر دنباله [1] X می‌نامیم اگر دنبالهٔ صعودی $\langle i_1, i_2, \cdots, i_k \rangle$ از اندیس‌های X وجود داشته باشند، به طوری‌که به ازای $j = 1, 2, \cdots, k$ داشته باشیم $x_{i_j} = z_j$.
- برای مثال دنبالهٔ $Z = \langle B, C, D, B \rangle$ یک زیردنباله از دنبالهٔ $X = \langle A, B, C, B, D, A, B \rangle$ است با اندیس‌های $\langle 2, 3, 5, 7 \rangle$

[1] subsequence

طولانی‌ترین زیر رشته مشترک

- به ازای دو دنبالهٔ X و Y ، می‌گوییم دنبالهٔ Z یک زیردنبالهٔ مشترک [1] X 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 زیردنبالهٔ مشترک X و Y است.
- دنبالهٔ $\langle B, C, A \rangle$ با طول ۳ طولانی‌ترین زیر دنبالهٔ مشترک X و Y نیست، چرا که دنبالهٔ $\langle B, C, B, A \rangle$ با طول ۴ وجود دارد که زیر دنبالهٔ مشترک X و Y است. این زیردنباله، طولانی‌ترین زیر دنبالهٔ مشترک [2] X و Y است، چرا که زیردنبالهٔ مشترک بلندتری وجود ندارد.
- می‌خواهیم این مسئله را به روش برنامه‌ریزی پویا حل کنیم.

[1] common subsequence
[2] longest common subsequence

طولانی‌ترین زیر رشته مشترک

- گام اول : مشخص کردن ساختار جواب مسئله
- می‌توانیم مسئله طولانی‌ترین زیردنبالهٔ مشترک را با استفاده از یک روش جستجوی کامل [1] 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.

[1] exhaustive search (brute-force search)

طولانی‌ترین زیر رشته مشترک

- برای تعریف مسئله طولانی‌ترین زیررشتهٔ مشترک با استفاده از زیر مسئله‌ها، ابتدا مفهوم پیشوند [1] یک دنباله را تعریف می‌کنیم.
- به ازای دنبالهٔ $X = \langle x_1, x_2, \cdots, x_m \rangle$، i امین پیشوند X برابراست با $X_i = \langle x_1, x_2, \cdots, x_i \rangle$ به ازای $i = 0, 1, \cdots, m$.
- 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 دنبالهٔ تهی است.

[1] prefix

طولانی‌ترین زیر رشته مشترک

- فرض کنید $X = \langle x_1, x_2, \cdots, x_m \rangle$ و $Y = \langle y_1, y_2, \cdots, y_n \rangle$ دو دنباله باشند و $Z = \langle z_1, z_2, \cdots, z_k \rangle$ 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

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

- (۱) 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 می توانیم 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 $z_k = x_m = y_n$. پس Z_{k-1} یعنی $k-1$ امین پیشوند Z باید یک زیررشته مشترک X_{n-1} و Y_{n-1} باشد. 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 $z_k \neq x_m$ 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) این اثبات شبیه و متقارن مورد (۲) است.

طولانی‌ترین زیر رشته مشترک

- بنابراین توانستیم مسئله طولانی‌ترین زیر رشته مشترک را بر اساس زیر مسئله‌های بهینه آن تعریف کنیم.

طولانی‌ترین زیر رشته مشترک

- فرض کنید $c[i,j]$ طول بلندترین زیررشتهٔ مشترک X_i و Y_j باشد.
- این مسئله بهینه‌سازی را می‌توانیم بر اساس زیر ساختارهای بهینه به صورت زیر تعریف کنیم.

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

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

طولانی‌ترین زیر رشته مشترک

- گام سوم : محاسبه طول طولانی‌ترین زیر دنباله مشترک
- از آنجایی که برای دو دنباله $X = \langle x_1, x_2, \cdots, x_m \rangle$ و $Y = \langle y_1, y_2, \cdots, y_n \rangle$ 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 برابراست با $\Theta(mn)$. طول زیر رشته مشترک برابراست با مقدار محاسبه شده برای $c[m, n]$.

طولانی‌ترین زیر رشته مشترک

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

Algorithm Longest Common Subsequence Length

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

طولانی‌ترین زیر رشته مشترک

Algorithm Longest Common Subsequence Length

```
function LCS-LENGTH(X,Y, m, n)
for i = 1 to m do  ▷ compute table entries in row-major order
   for j = 1 to n do
      if X[i] == Y[j] then
         c[i, j] = c[i-1, j-1] + 1
         b[i, j] = "↖"
      else if c[i-1, j] ⩾ c[i, j-1] then
         c[i, j] = c[i-1, j]
         b[i, j] = "↑"
      else
         c[i, j] = c[i, j-1]
         b[i, j] = "←"
return c and b
```

طولانی‌ترین زیر رشته مشترک

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

طولانی‌ترین زیر رشته مشترک

Algorithm Print Longest Common Subsequence

```
function PRINT-LCS(b, X, i, j)
if i == 0 or j ==0 then
    return      ▷ the longest common subsequence has length 0
if b[i, j] == "↖" then
    Print-LCS(b, X, i-1, j-1)
    print X[i]      ▷ same as Y[j]
else if b[i, j] = "↑" then
    Print-LCS(b, X, i-1, j)
else
    Print-LCS(b, X, i, j-1)
```

طولانی‌ترین زیر رشته مشترک

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

i	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

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 دنبالهٔ مشترک به طور مثال می توانیم جدول b را حذف کنیم و اطلاعات لازم برای ساختن بلندترین زیر دنبالهٔ مشترک را از جدول c به دست آوریم.
- هریک از درایه‌های $c[i, j]$ از طریق یکی از سه درایهٔ $c[i-1, j-1]$ ، $c[i-1, j]$ ، $c[i, j-1]$ محاسبه شده است که در زمان ثابت می‌توانیم بدون جدول b به دست آوریم درایه $c[i, j]$ چگونه محاسبه شده است.
- بنابراین طولانی‌ترین زیردنبالهٔ مشترک را می‌توانیم همچنان در زمان $\Theta(m+n)$ بسازیم و جدول b را حذف کرده و از حافظهٔ 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 کلمهٔ 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 کلمهٔ انگلیسی و معادل فارسی آن را شامل شود.
- اگر از یک درخت جستجوی دودویی متوازن [1] 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

درخت جستجوی دودویی بهینه

- 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 فاصلهٔ آنها از ریشه بیشتر باشد.
- با استفاده از درخت جستجوی دودویی بهینه [1] می‌توان کلمات را به گونه‌ای ذخیره و بازیابی کرد که کلمات با احتمال وقوع بیشتر نزدیک‌تر به ریشه قرار بگیرند.

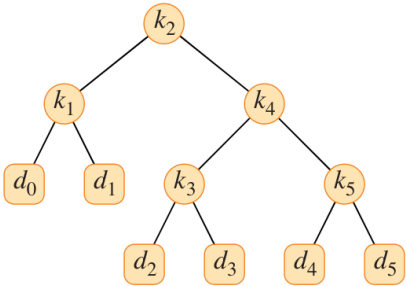
[1] optimal binary search tree

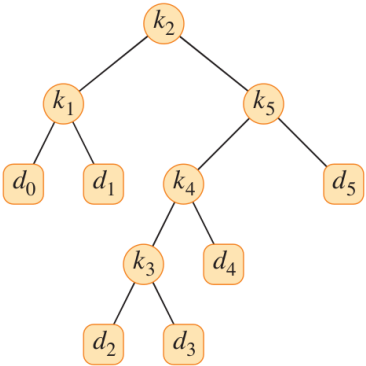
درخت جستجوی دودویی بهینه

- دنبالهٔ $K = \langle k_1, k_2, \cdots, k_n \rangle$ با n کلید را در نظر بگیرید به طوری‌که $k_1 < k_2 < \cdots < k_n$.
- 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 $d_0, d_1, d_2, \cdots, d_n$ نیز داریم که نماینده این کلیدها هستند. در واقع d_0 نمایندهٔ همهٔ کلیدهایی است که از k_1 کوچکترند و d_n نمایندهٔ همهٔ کلیدهایی است که از k_n بزرگ‌ترند و همچنین به ازای $i = 1, 2, \cdots, n - 1$ ، کلید d_i نمایندهٔ همهٔ 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 بی‌استفادهٔ 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 بی‌استفادهٔ 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 هزینهٔ جستجوی یک کلید در درخت به ازای هر بار جستجو برابر با تعداد رئوس بررسی شده برای رسیدن به آن کلید باشد. بنابراین هزینهٔ جستجوی یک کلید در یک جستجو برابر خواهد بود با عمق [1] رأس مربوط به آن کلید به علاوهٔ یک. ریشه در عمق صفر قرار دارد، بنابراین هزینهٔ یافتن کلید مربوط به ریشه در یک جستجو برابر است با یک.
- برای یافتن هزینهٔ جستجوی یک کلید در یک متن، باید هزینهٔ یک بار جستجو را در احتمال وقوع آن کلید ضرب کنیم.
- نهایتا برای یافتن هزینهٔ جستجوی یک درخت باید هزینهٔ جستجوی همهٔ کلیدها را با هم جمع کنیم.

[1] depth

درخت جستجوی دودویی بهینه

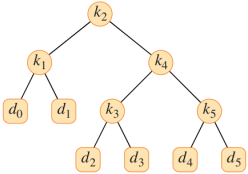
- بنابراین هزینهٔ جستجو در درخت T برابر است با :

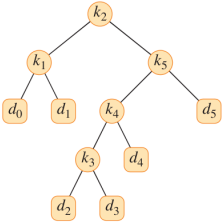
$$
\begin{aligned}
E[\text{search cost in } T] &= \sum_{i=1}^{n} (\mathrm{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\mathrm{depth}_T(d_i) + 1) \cdot q_i \\
&= 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \mathrm{depth}_T(d_i) \cdot q_i
\end{aligned}
$$

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

درخت جستجوی دودویی بهینه

- در شکل زیر هزینهٔ 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

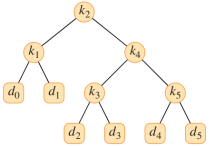
درخت جستجوی دودویی بهینه

- حال به ازای تعدادی کلید به همراه احتمال وقوع آنها، می‌خواهیم یک درخت جستجوی دودویی بیابیم که هزینهٔ جستجو در آن حداقل است.
- به این درخت، درخت جستجوی دودویی بهینه [1] گفته می‌شود.

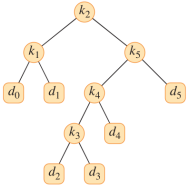
[1] optimal binary search tree

درخت جستجوی دودویی بهینه

- در شکل زیر دو درخت جستجوی دودویی نشان داده شده‌اند. هزینهٔ 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 همهٔ کلیدها با احتمال وقوع بیشتر در ریشهٔ آن باشند. برای مثال کلید k_5 بیشترین احتمال وقوع را دارد، اما ریشهٔ درخت جستجو k_2 است.

درخت جستجوی دودویی بهینه

- همانند مسئله ضرب زنجیره‌ای ماتریس‌ها، با استفاده از جستجوی کامل برای بررسی همهٔ درخت‌های جستجو نمی‌توانیم در زمان معقول درخت جستجوی دودویی بهینه را به دست آوریم. تعداد همهٔ درخت‌های جستجوی دودویی از مرتبهٔ نمایی است، بنابراین بررسی همهٔ 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.
- یک زیردرخت دلخواه را در نظر بگیرید. این زیردرخت شامل کلیدهای $k_i, \cdots, k_j$ است که برگ‌های آن را کلیدهای $d_{i-1}, \cdots, d_j$ تشکیل می‌دهند، به طوری‌که $1 \leqslant i \leqslant j \leqslant n$
- اگر یک درخت جستجوی دودویی بهینه T داشته باشیم، زیر درخت T' شامل کلیدهای $k_i, \cdots, k_j$ و کلیدهای $d_{i-1}, \cdots, d_j$ نیز باید بهینه باشد. اگر یک زیر درخت T'' وجود داشت که هزینهٔ آن کمتر از T' بود، می‌توانستیم T' را با T'' جایگزین کنیم و یک درخت بهینه‌تر به جای T بیابیم که با فرض بهینه بودن T در تناقض است.
- حال می‌خواهیم مسئله را با استفاده از زیر مسئله‌های بهینهٔ آن حل کنیم.

درخت جستجوی دودویی بهینه

- به ازای کلیدهای $k_i, \cdots, k_j$ ، یکی از این کلیدها، برای مثال کلید k_r به طوری‌که $i \leqslant r \leqslant j$ ، ریشهٔ زیردرخت بهینه برای این کلیدهاست.
- زیردرخت سمت چپ ریشهٔ k_r شامل کلیدهای $k_i, \cdots, k_{r-1}$ و کلیدهای برگ $d_{i-1}, \cdots, d_{r-1}$ است و زیردرخت سمت راست شامل کلیدهای $k_{r+1}, \cdots, k_j$ و کلیدهای برگ $d_r, \cdots, d_j$ است.
- فرض کنید در یک زیردرخت با کلیدهای $k_i, \cdots, k_j$ ، کلید k_i را به عنوان ریشه انتخاب کنیم. بنابراین زیردرخت سمت چپ این زیردرخت هیچ رأسی را شامل نمی‌شود. اگر برگ‌ها را نیز در نظر بگیریم، 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 $k_i, \cdots, k_j$ را در نظر بگیرید به طوری‌که $i \geqslant 1$ ، $j \leqslant n$ و $j \geqslant i-1$. وقتی $j = i-1$ باشد، تنها کلید برگ d_{i-1} را خواهیم داشت.
- فرض کنید $e[i,j]$ هزینهٔ جستجوی یک درخت جستجوی بهینه با کلیدهای $k_i, \cdots, k_j$ باشد. هدف محاسبهٔ هزینه جستجو برای همهٔ کلیدهاست که برابر با مقدار $e[1,n]$ می‌باشد.
- اگر $j = i-1$ باشد، آنگاه مسئله تنها شامل یک کلید d_{i-1} می‌شود. در اینصورت هزینهٔ جستجو برابراست با $e[i,i-1] = q_{i-1}$.
- وقتی $j \geqslant i$ باشد، باید ریشهٔ k_r را از بین کلیدهای $k_i, \cdots, k_j$ انتخاب کنیم و یک درخت جستجوی بهینه با کلیدهای $k_i, \cdots, k_{r-1}$ به عنوان زیردرخت سمت چپ ریشهٔ k_r و یک درخت جستجوی بهینه با کلیدهای $k_{r+1}, \cdots, k_j$ به عنوان زیردرخت سمت راست ریشهٔ 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 هزینهٔ جستجو برای رئوس زیردرخت T' در درخت T به میزان مجموع احتمال رئوس T' افزایش می‌یابد.
- برای یک زیردرخت با کلیدهای $k_i, \cdots, k_j$ 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 ریشهٔ یک زیردرخت بهینه با کلیدهای $k_i, \cdots, k_j$ 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 ریشهٔ k_r انتخاب می‌شود.
- از آنجایی که هدف این است که ریشه‌ای را انتخاب کنیم که مقدار هزینه جستجو را کاهش دهد، بنابراین رابطهٔ بازگشتی برای محاسبهٔ هزینه جستجو در درخت بهینه را به صورت زیر می‌نویسیم.

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

- 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 $1 \leqslant i \leqslant j \leqslant n$ برابراست با اندیس r برای کلید k_r که ریشهٔ درخت جستجوی بهینه‌ای است که برای کلیدهای $k_i, \cdots, k_j$ یافته می‌شود.

درخت جستجوی دودویی بهینه

- گام سوم : محاسبهٔ هزینهٔ 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.
- همهٔ مقادیر $e[i, j]$ به ازای $j \geqslant i - 1$ باید محاسبه شوند. جدول $root[i, j]$ ریشهٔ زیردرخت‌ها را با کلیدهای $k_i, \cdots, k_j$ ذخیره می‌کند، به طوری‌که $1 \leqslant i \leqslant j \leqslant n$.

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 $1 \leqslant i \leqslant n+1$
- به ازای $j \geqslant i$ 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.

درخت جستجوی دودویی بهینه

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

Algorithm Optimal-BST

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

درخت جستجوی دودویی بهینه

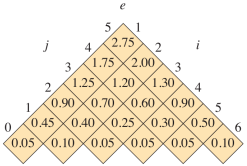
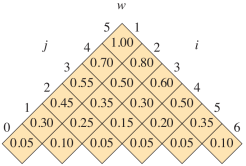
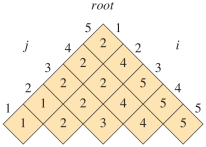
Algorithm Optimal-BST

```
function OPTIMAL-BST(p, q, n)
for t = 1 to n do
    for i = 1 to n - t + 1 do
        j = i + t - 1
        e[i,j] = ∞
        w[i,j] = w[i,j-1] + p[j] + q[j]
        for r = i to j do          ▷ try all possible roots r
            t = e[i,r-1] + e[r+1,j] + w[i,j]
            if t < e[i,j] then          ▷ new minimum?
                e[i,j] = t
                root[i,j] = r
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 اندازهٔ $\Theta(n^2)$ نیاز دارد.

درخت جستجوی دودویی بهینه

- در شکل زیر، جدول‌های $e[i,j]$ ، $w[i,j]$ و $\mathrm{root}[i,j]$ 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