دانشگاه کردستان
University of Kurdistan

طراحی و تحلیل الگوریتم ها

مدرس: سعدون عزیزی

s.azizi@uok.ac.ir

گروه مهندسی کامپیوتر

نیم سال دوم ۹۷-۹۶

تکنیک‌های طراحی الگوریتم‌ها

- تقسیم و حل
- **برنامه‌ریزی پویا**
- الگوریتم‌های حریصانه
- الگوریتم‌های عقبگرد
- الگوریتم‌های شاخه و حد

برنامه‌ریزی پویا

- الگوریتم‌های تقسیم و حل مسئله را به زیرمسئله‌های مستقل تقسیم کرده، آنها را به صورت بازگشتی حل نموده و سپس جواب زیرمسئله را با هم ترکیب می‌کنند تا جواب مسئله کلی به دست آید.

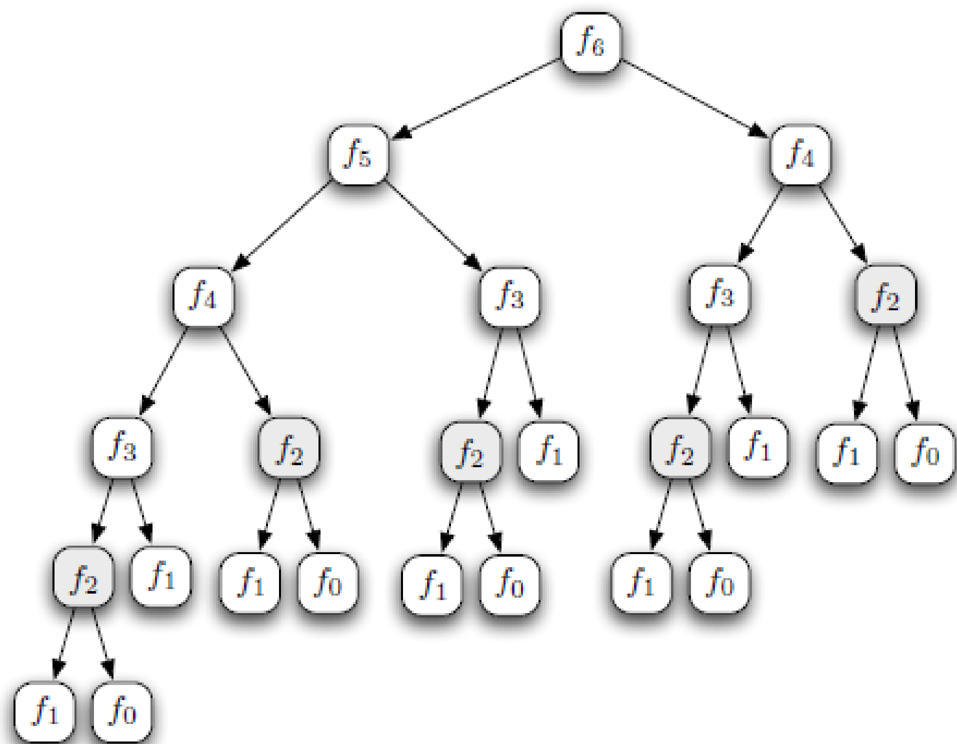
- در مسائلی که زیرمسئله‌ها با هم همپوشانی دارند (زیرمسئله‌های مشترک)، الگوریتم‌های تقسیم و حل زیرمسئله‌ها را بیشتر از یکبار حل می‌کنند که این منجر به کارایی بسیار پایین می‌شود.

- یک الگوریتم برنامه‌ریزی پویا هر زیر مسئله را فقط یکبار حل، و جواب آن را ذخیره می‌کند تا در صورت نیاز، به جای محاسبه دوباره، آن را فقط بازیابی کند.

- برنامه‌ریزی پویا معمولا برای حل مسائل بهینه‌سازی به کار می‌رود.

دنباله فیبوناچی (روش تقسیم و حل)

$$f(n) = \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f(n-1) + f(n-2) & if\ n \geq 2 \end{cases}$$



- پیچیدگی این روش از مرتبه نمایی است (چرا؟)

دنباله فیبوناچی (روش برنامه‌ریزی پویا)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610

```
Fibo(n) {
int A[0..n], i;
A[0]=0;
A[1]=1;
for (i=2; i<=n; i++)
        A[i]=A[i-1]+A[i-2];
return A[n];
}
```

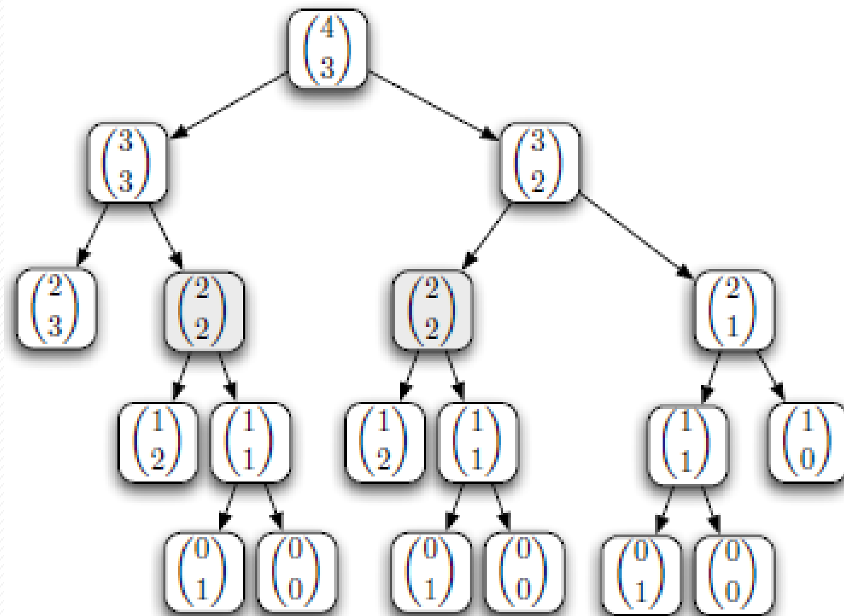
پیچیدگی زمانی الگوریتم: $O(n)$

پیچیدگی حافظه: $O(n)$

سوال: آیا می‌توان پیچیدگی حافظه را کاهش داد؟

انتخاب k شیء از n شیء (روش تقسیم و حل)

$$\binom{n}{k} = \begin{cases} 0 & \text{if } n < k \\ 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{otherwise} \end{cases}$$



انتخاب k شیء از n شیء (روش برنامه‌ریزی پویا)

$$B[i][j] = \begin{cases} 0 & \text{if } j < i \\ 1 & \text{if } j = 0 \text{ or } j = i \\ B[i-1][j] + B[i-1][j-1] & 0 < \mathrm{j} < \mathrm{i} \end{cases}$$

j

i		0	1	2	3	4
	0	1	0	0	0	0
	1	1	1	0	0	0
	2	1	2	1	0	0
	3	1	3	3	1	0
	4	1	4	6	4	1
	5	1	5	10	10	5
	6	1	6	15	20	15
	7	1	7	21	35	35
	8	1	8	28	56	70

انتخاب k شیء از n شیء (روش برنامه‌ریزی پویا)

$$B[i][j] = \begin{cases} 0 & \text{if } j < i \\ 1 & \text{if } j = 0 \text{ or } j = i \\ B[i-1][j] + B[i-1][j-1] & 0 < \text{j} < \text{i} \end{cases}$$

```
Choose(k,n) {
  int i,j,B[0..n][0..k];
  for(i=0; i<=n; i++)
    for(j=0; j<=min(i,k); j++)
      if(j==0 || j==i)
        B[i][j]=1;
      else
        B[i][j]=B[i-1][j]+B[i-1][j-1];
  return B[n][k]; }
```

پیچیدگی زمانی الگوریتم: $O(nk)$

پیچیدگی حافظه: $O(nk)$

سوال: آیا می‌توان پیچیدگی حافظه را کاهش داد؟

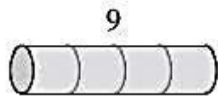
مسئله برش میله

- موسسه‌ای میله‌های فولادی به اندازه n را خریداری کرده و می‌خواهد آنها را برش دهد و سپس قطعه‌های کوچکتر را بفروشد.
- قیمت هر قطعه معلوم است.
- هزینه برش‌ها ناچیز (قابل چشم‌پوشی) است.
- این موسسه می‌خواهد بداند که برش‌ها به چه صورتی باشد تا بیشترین سود را به دست آورد.

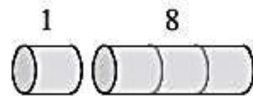
مسئله برش میله

- مثال:

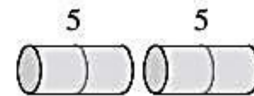
طول قطعه‌ی i	1	2	3	4	5	6	7	8	9	10
قیمت p_i	1	5	8	9	10	17	17	20	24	30



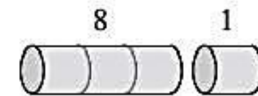
(a)



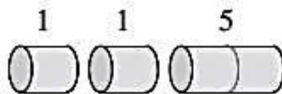
(b)



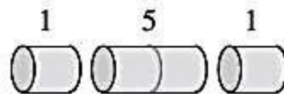
(c)



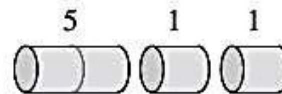
(d)



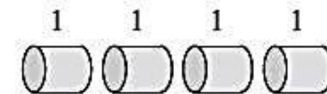
(e)



(f)



(g)



(h)

مسئله برش میله

- **سوال:** یک میله به طول n را به چند روش مختلف می‌توان برش داد؟
- **پاسخ:** 2^{n-1} چرا؟
- مسئله برش میله دارای زیرساختار بهینه (optimal substructure) است: جواب‌های بهینه به یک مسئله از جواب‌های بهینه زیرمسئله‌ها تشکیل می‌شود.

تعریف

گفته می‌شود اصل بهینگی در یک مسئله صدق می‌کند اگر حل بهینه برای آن مسئله، همواره شامل حل بهینه برای همه‌ی زیرمسئله‌های آن باشد.

مسئله برش میله

- ارائه رابطه بازگشتی برای مسئله:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

p_i : قیمت قطعه به طول i

r_n : بیشترین سود حاصل از برش میله

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = −∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n − i))
6  return q
```

مسئله برش میله

- تحلیل زمان اجرای الگوریتم CUT-ROD

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

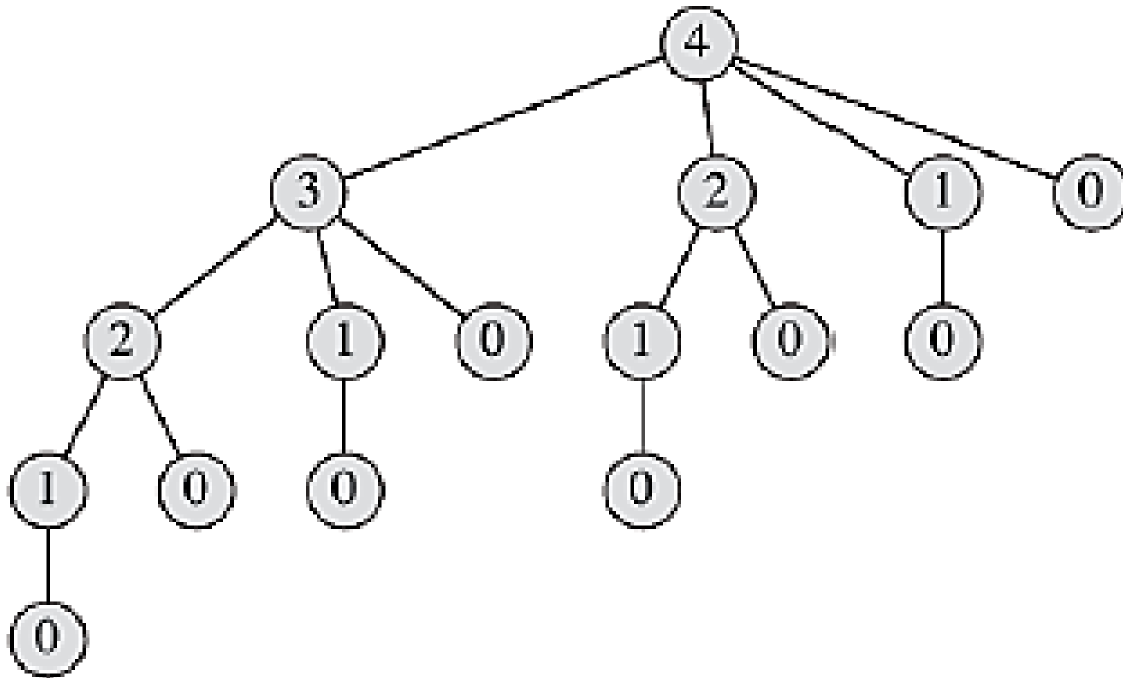
- با حل رابطه بازگشتی بالا، داریم:

$$T(n) = 2^n$$

- چرا الگوریتم CUT-ROD این قدر ناکارامد است؟

مسئله برش میله

- درخت بازگشتی حاصل از فراخوانی CUR-ROD(p,n) برای n=4



مسئله برش میله

- استفاده از برنامه‌ریزی پویا

- رویکرد بالا به پایین با به‌خاطرسپاری (top-down with memoization)

  - در این روش، مانند روش معمولی رویه را به صورت بازگشتی می‌نویسیم، ولی آن را طوری اصلاح می‌کنیم که نتیجه هر زیرمسئله را ذخیره کند. بنابراین، قبل از حل هر زیرمسئله‌ای ابتدا چک می‌کند که آیا این زیرمسئله را قبلا حل کرده است یا نه. اگر قبلا حل شده بود، مقدار ذخیره شده را باز می‌گرداند.

- رویکرد پایین به بالا (bottom-up method)

در این روش، ابتدا کوچک‌ترین زیرمسئله‌ها را حل می‌کنیم و جواب آنها را ذخیره می‌نماییم. سپس هرجا به جواب این زیرمسئله‌ها نیاز باشد کافی است که جواب آنها را بازیابی کنیم.

- در هر دو حالت، هر زیر مسئله را فقط یک بار حل می‌کنیم.

مسئله برش میله

- رویکرد بالا به پایین با به‌خاطرسپاری

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

مسئله برش میله

- رویکرد پایین به بالا

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

مسئله برش میله

- زمان اجرای الگوریتم MEMOIZED-CUT-ROD

$\theta(n^2)$

- زمان اجرای الگوریتم BOTTOM-UP-CUT-ROD

$\theta(n^2)$

مسئله برش میله

- بازسازی یک جواب

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] and s[0..n] be new arrays
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          if q < p[i] + r[j - i]
7              q = p[i] + r[j - i]
8              s[j] = i
9      r[j] = q
10 return r and s
```

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

- برای $n=10$ فقط عدد ۱۰ چاپ می‌شود
- برای $n=7$ اعداد ۱ و ۶ چاپ می‌شوند

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n - s[n]
```

مسئله ضرب زنجیره‌ی ماتریس‌ها

- فرض کنید که یک زنجیره (دنباله) از n ماتریس داشته باشیم و بخواهیم که آنها را در یکدیگر ضرب کنیم؛ یعنی

$$A = A_1 \times A_2 \times \cdots \times A_n$$

جایی که ماتریس A_i دارای p_{i-1} سطر و p_i ستون است.

- هدف مسئله این است که ضرب $A_n \times \cdots \times A_2 \times A_1$ را طوری پرانتزگذاری کنید که تعداد ضرب‌های اسکالر کمینه شود.

- **سوال:** تعداد پرانتزگذاری‌های ممکن چقدر است؟

- **پاسخ:** عدد کاتالان $C_n = \frac{1}{n+1}\binom{2n}{n}$ توجه: در اینجا منظور از n تعداد ضرب‌های بین ماتریس‌ها است.

n	0	1	2	3	4	5	6
C_n	1	1	2	5	14	42	128

مسئله ضرب زنجیره‌ی ماتریس‌ها

- **مثال:** زنجیره‌ی $A = A_1 \times A_2 \times A_3 \times A_4$ را در نظر بگیرید که در آن $p_0 = 10$، $p_1 = 20$، $p_2 = 50$، $p_3 = 1$ و $p_4 = 100$ است.

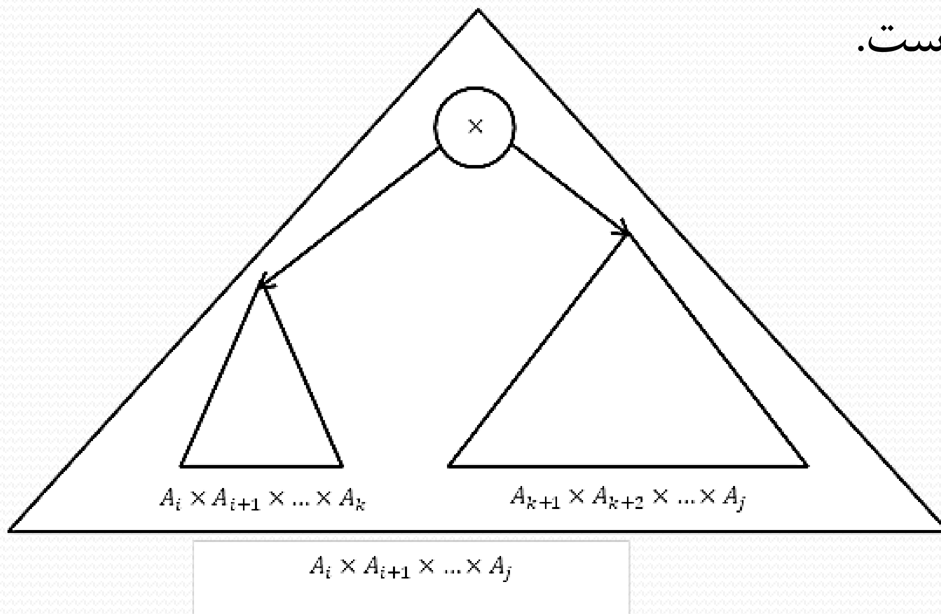
نوع پرانتزگذاری	تعداد ضرب‌های اسکالر
$((A_1 \times A_2) \times$	11500
$(A_1 \times (A_2 \times A$	2200
$(A_1 \times A_2) \times ($	15000
$A_1 \times ((A_2 \times A$	23000
$A_1 \times (A_2 \times ($	125000

مسئله ضرب زنجیره‌ی ماتریس‌ها

- ارائه رابطه بازگشتی برای حل مسئله به روش برنامه‌ریزی پویا:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\,, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j\,. \end{cases}$$

جایی که m[i,j] کمینه تعداد ضرب‌های اسکالر مورد نیاز برای محاسبه حاصل ضرب ماتریس‌های $A_i \times A_{i+1} \times \cdots \times A_j$ است.



مسئله ضرب زنجیره‌ی ماتریس‌ها

```
MATRIX-CHAIN-ORDER(p)
1   n = p.length - 1
2   let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
3   for i = 1 to n
4       m[i, i] = 0
5   for l = 2 to n              // l is the chain length
6       for i = 1 to n - l + 1
7           j = i + l - 1
8           m[i, j] = ∞
9           for k = i to j - 1
10              q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

- s[i,j]: برای ذخیره کردن مقدار k بهینه پیدا شده
- پیچیدگی زمان اجرا: $O(n^3)$
- پیچیدگی حافظه: $O(n^2)$

مسئله ضرب زنجیره‌ی ماتریس‌ها

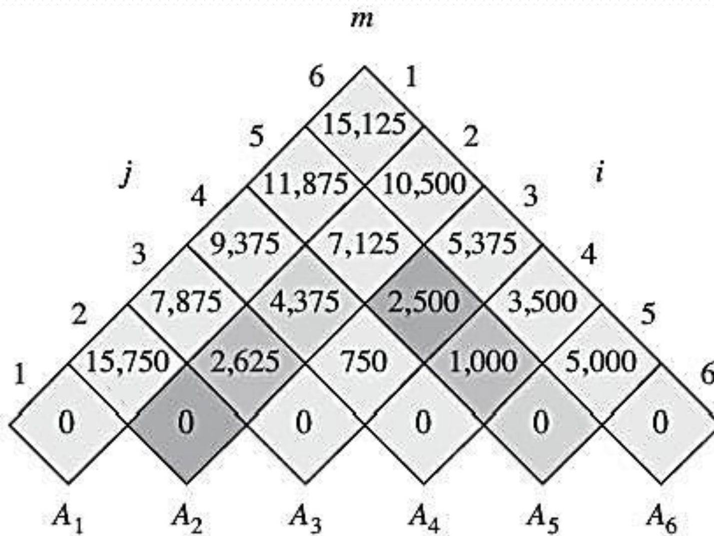
- ساختن یک جواب بهینه

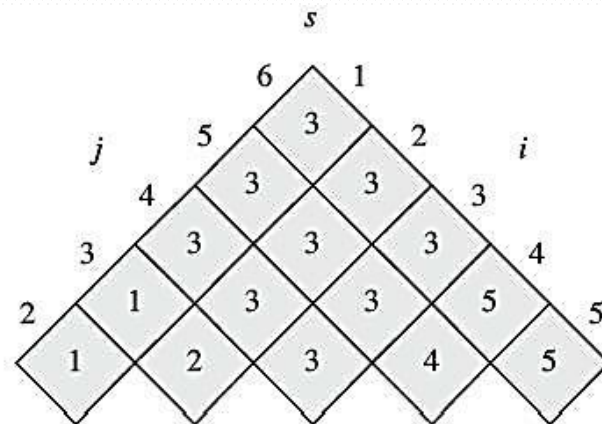
```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A"_i
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6      print ")"
```

مسئله ضرب زنجیره‌ی ماتریس‌ها

- مثال:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25





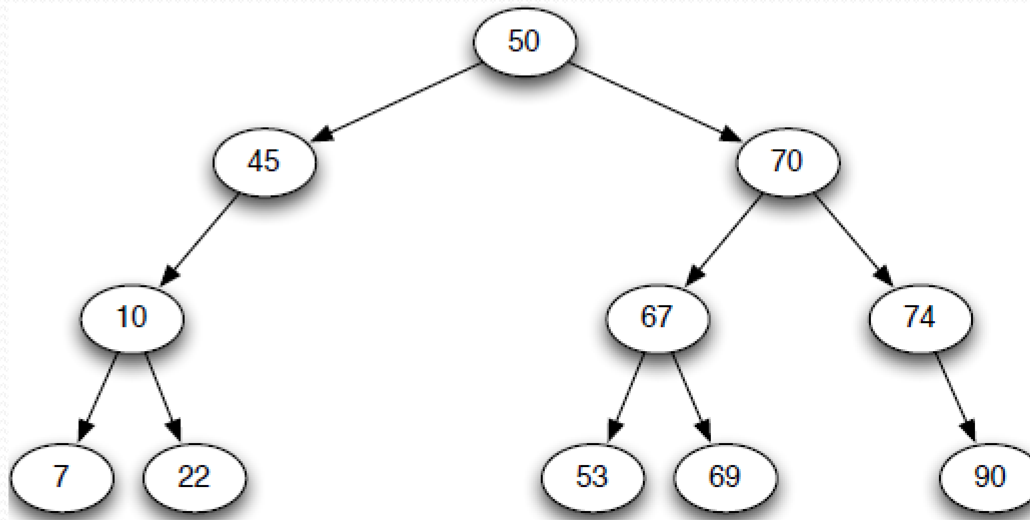
$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000 , \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 , \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$
$$= 7125 .$$

درخت‌های جستجوی دودویی بهینه

- درخت جستجوی دودویی (Binary Search Tree) درختی است که در آن مقدار هر گره از گره سمت چپ خود بیشتر و از گره سمت راست کمتر است.

- پیمایش میانوندی هر BST لیست مرتب شده‌ای به ما می‌دهد.

- عمق هر درخت جستجوی دودویی برابر است با: $\log(n) \leq depth(BST) \leq n$



درخت‌های جستجوی دودویی بهینه

- فرض کنید که یک دنباله $\mathrm{K} = (k_1, k_2, \ldots, k_n)$ از n کلید مجزا به صورت مرتب شده (به طوری که $k_1 < k_2 < \cdots < k_n$) به ما داده شده است.
- برای هر کلید k_i یک احتمال p_i داریم که احتمال جستجو برای k_i را نشان می‌دهد.
- ممکن است بعضی جستجوها برای مقادیری انجام شود که در K موجود نباشند؛ بنابراین $n+1$ کلید زائد $d_0, d_1, d_2, \ldots, d_n$ داریم که نشان‌دهنده مقادیری هستند که در K وجود ندارند.
- کلید زائد d_0 نشان‌دهنده تمام مقادیر کوچکتر از k_1، کلید زائد d_n نشان‌دهنده تمام مقادیر بزرگتر از k_n و برای $i=1,2,\ldots,n-1$، کلید زائد d_i نشان‌دهنده تمام مقادیر بین k_i و k_{i+1} است.
- برای هر کلید زائد d_i نیز یک احتمال q_i برای جستجو داریم.

درخت‌های جستجوی دودویی بهینه

- هر کلید k_i یک گره داخلی و هر کلید زائد d_i یک برگ است.
- هر جستجو یا موفق است (یافتن کلید k_i) و یا ناموفق (یافتن یک کلید زائد d_i)، بنابراین داریم: $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$
- امید ریاضی هزینه یک جستجو در درخت T برابر است با:

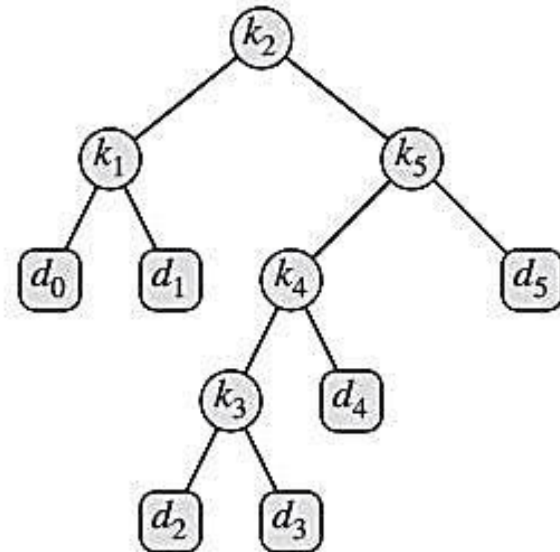
$$\begin{aligned} \mathrm{E}\,[\text{search cost in } T] &= \sum_{i=1}^{n} (\mathrm{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\mathrm{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \mathrm{depth}_T(d_i) \cdot q_i \ , \end{aligned}$$

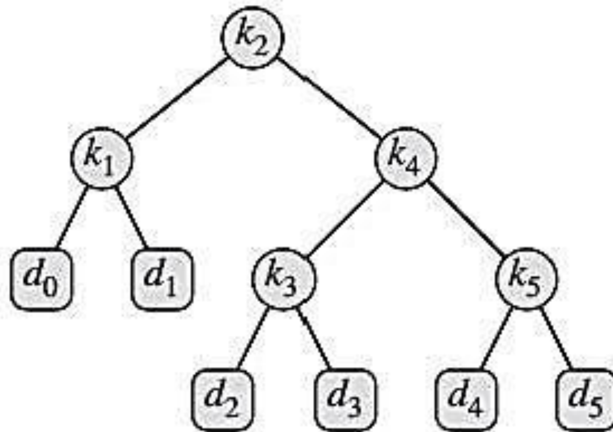
- هدف ساختن یک درخت جستجوی دودویی بهینه است. یعنی امید ریاضی هزینه جستجو در آن کمینه باشد.

درخت‌های جستجوی دودویی بهینه

- مثال:

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





- هزینه درخت سمت چپ برابر است با ۲/۸۰، هزینه درخت سمت راست برابر است با ۲/۷۵

درخت‌های جستجوی دودویی بهینه

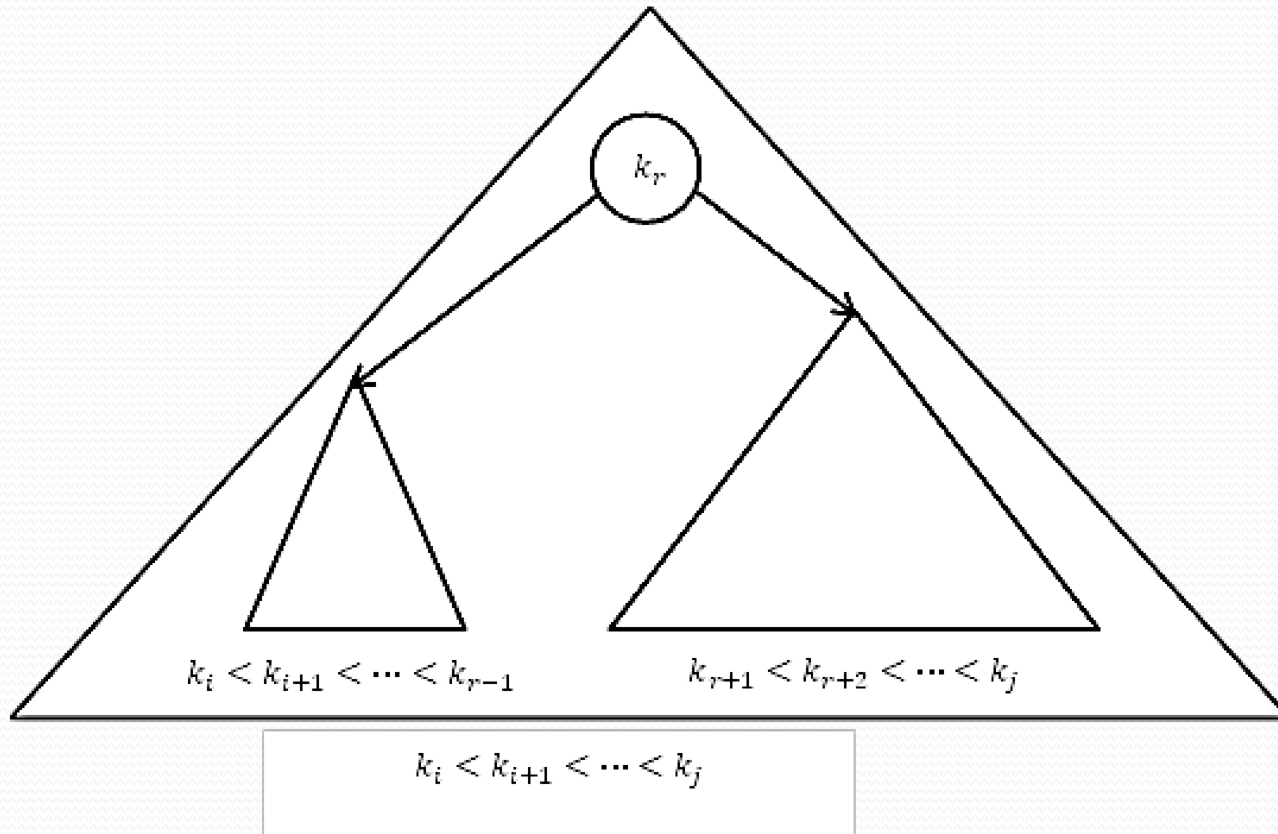
- چگونه درخت جستجوی دودویی بهینه را پیدا کنیم؟

- تعداد کل درخت‌های جستجوی دودویی با n گره برابر است با عدد کاتالان $C_n = \frac{1}{n+1}\binom{2n}{n}$

- یعنی $\Omega\left({4^n}/{n^{\frac{3}{2}}}\right)$

- روش بهتر؟ استفاده از برنامه‌ریزی پویا

درخت‌های جستجوی دودویی بهینه



درخت‌های جستجوی دودویی بهینه

- تعریف می‌کنیم e[i,j] برابر است با امید ریاضی هزینه جستجو در درخت جستجوی دودویی بهینه حاوی کلیدهای $k_i, \ldots, k_j$.
- اگر j=i-1 باشد، در این صورت فقط کلید زائد d_{i-1} را داریم که امید ریاضی هزینه جستجوی آن $e[i,i-1] = q_{i-1}$ است.
- وقتی $j \geq i$ باشد، باید یک ریشه k_r از میان کلیدهای $k_i, \ldots, k_j$ انتخاب کنیم و سپس یک درخت جستجوی دودویی بهینه با کلیدهای $k_i, \ldots, k_{r-1}$ به عنوان زیردرخت سمت چپ و یک درخت جستجوی دودویی بهینه با کلیدهای $k_{r+1}, \ldots, k_j$ به عنوان زیر درخت سمت راست بسازیم.
- $w(i,j)$ را به صورت زیر تعریف می‌کنیم:

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

درخت‌های جستجوی دودویی بهینه

- اگر k_r ریشه درخت جستجوی دودویی بهینه حاوی کلیدهای $k_i, \dots, k_j$ باشد، داریم

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

- با توجه به این که

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

- e[i,j] را به صورت زیر بازنویسی می‌کنیم

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

- رابطه بازگشتی نهایی:

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1\,, \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j\,. \end{cases}$$

درخت‌های جستجوی دودویی بهینه

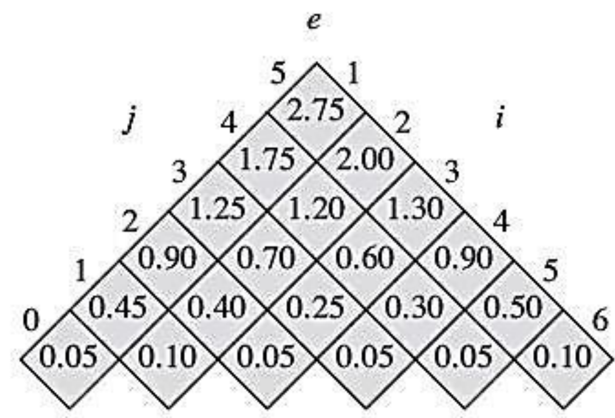
```
OPTIMAL-BST(p, q, n)
 1  let e[1..n+1, 0..n], w[1..n+1, 0..n],
            and root[1..n, 1..n] be new tables
 2  for i = 1 to n+1
 3      e[i, i-1] = q_{i-1}
 4      w[i, i-1] = q_{i-1}
 5  for l = 1 to n
 6      for i = 1 to n-l+1
 7          j = i+l-1
 8          e[i, j] = ∞
 9          w[i, j] = w[i, j-1] + p_j + q_j
10          for r = i to j
11              t = e[i, r-1] + e[r+1, j] + w[i, j]
12              if t < e[i, j]
13                  e[i, j] = t
14                  root[i, j] = r
15  return e and root
```

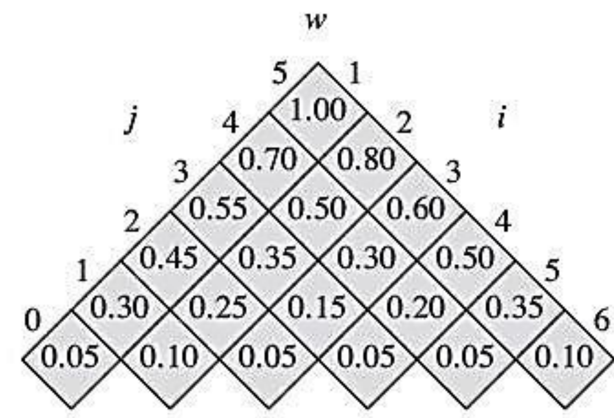
- root[i,j]: برای ذخیره کردن مقدار r بهینه پیدا شده
- پیچیدگی زمان اجرا: $O(n^3)$
- پیچیدگی حافظه: $O(n^2)$

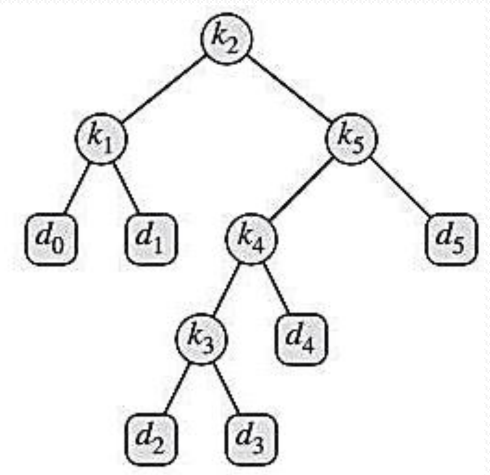
درخت‌های جستجوی دودویی بهینه

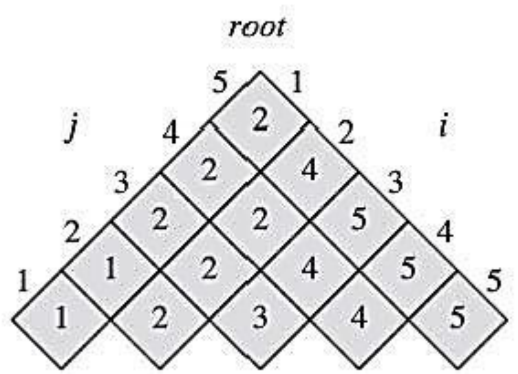
- مثال:

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









طولانی‌ترین زیردنباله مشترک

تعریف

با داشتن دنباله $X=(x_1,x_2,\ldots,x_m)$، دنباله $Z=(z_1,z_2,\ldots,z_k)$ یک زیردنباله از X است اگر یک دنباله صعودی اکید $(i_1,i_2,\ldots,i_k)$ از اندیس‌های X موجود باشد به طوری که برای تمام j=1,2,…,k، داشته باشیم $x_{i_j}=z_i$.

- **مثال:** Z=(B,C,D,B) یک زیردنباله از X=(A,B,C,B,D,A,B) با اندیس‌های مربوطه (2,3,5,7) است.

طولانی‌ترین زیردنباله مشترک

تعریف

با داشتن دو دنباله X و Y، می‌گوییم دنباله Z یک زیردنباله مشترک X و Y است اگر Z هم زیردنباله‌ای از X و هم زیردنباله‌ای از Y باشد.

- **مثال:** Z=(B,C,A) یک زیردنباله مشترک برای X=(A,B,C,B,D,A,B) و Y=(B,D,C,A,B,A) است.

طولانی‌ترین زیردنباله مشترک

تعریف

با داشتن دنباله $X = (x_1, x_2, \ldots, x_m)$، iامین پیشوند X را برای $i=0,1,\ldots,m$ به صورت $X_i = (x_1, x_2, \ldots, x_i)$ تعریف می‌کنیم.

- **مثال:** اگر $X = (A, B, C, B, D, A, B)$ باشد، آنگاه $X_4 = (A, B, C, B)$ است.

طولانی‌ترین زیردنباله مشترک

- در مسئله بلندترین زیردنباله مشترک، ما دو دنباله $X = (x_1, x_2, \ldots, x_m)$ و $Y = (y_1, y_2, \ldots, y_n)$ را داریم و می‌خواهیم طولانی‌ترین زیردنباله مشترک X و Y را پیدا کنیم.

- **مثال:** طولانی‌ترین زیردنباله مشترک برای X=(A,B,C,B,D,A,B) و Y=(B,D,C,A,B,A) برابر است با:

Z=(B,C,B,A)

- یک رویکرد ساده و بدیهی برای حل این مسئله این است که تمام زیردنباله‌های X را بسازیم و سپس بررسی کنیم که آیا هر کدام از آنها زیردنباله Y هم هستند یا خیر و بلندترین زیردنباله مشترک پیدا شده را ذخیره کنیم.

- پیچیدگی این روش از مرتبه نمایی است!

طولانی‌ترین زیردنباله مشترک

- **قضیه:** فرض کنید $X = (x_1, x_2, \dots, x_m)$ و $Y = (y_1, y_2, \dots, y_n)$ دو دنباله باشند و $Z = (z_1, z_2, \dots, z_k)$ یک بلندترین زیردنباله مشترک از X و Y باشد.

1. اگر $x_m = y_n$، آنگاه $z_k = x_m = y_n$ و Z_{k-1} یک زیردنباله مشترک X_{m-1} و Y_{n-1} است.
2. اگر $x_m \neq y_n$، آنگاه $z_k \neq x_m$ نتیجه می‌دهد که Z یک زیردنباله مشترک از X_{m-1} و Y است.
3. اگر $x_m \neq y_n$، آنگاه $z_k \neq y_n$ نتیجه می‌دهد که Z یک زیردنباله مشترک از X و Y_{n-1} است.

- اثبات بر اساس برهان خلف.

طولانی‌ترین زیردنباله مشترک

- فرض کنید c[i,j] طول بلندترین زیردنباله مشترک دنباله‌های X_i و Y_j باشد.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \,. \end{cases}$$

- به کمک ماتریس زیر می‌توان جواب مسئله را ساخت.

$$B[i,j] = \begin{cases} \leftarrow \text{ or } \uparrow & \text{if } i = 0 \text{ or } j = 0, \\ \nwarrow & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ \leftarrow \text{ or } \uparrow & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

طولانی‌ترین زیردنباله مشترک

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i - 1, j - 1] + 1
12              b[i, j] = "↖"
13          elseif c[i - 1, j] ≥ c[i, j - 1]
14              c[i, j] = c[i - 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j - 1]
17              b[i, j] = "←"
18  return c and b
```

- زمان اجرای الگوریتم: $\theta(mn)$

طولانی‌ترین زیردنباله مشترک

- مثال: پیدا کردن بلندترین زیردنباله مشترک دو دنباله X=(A,B,C,B,D,A,B) و Y=(B,D,C,A,B,A)

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

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```