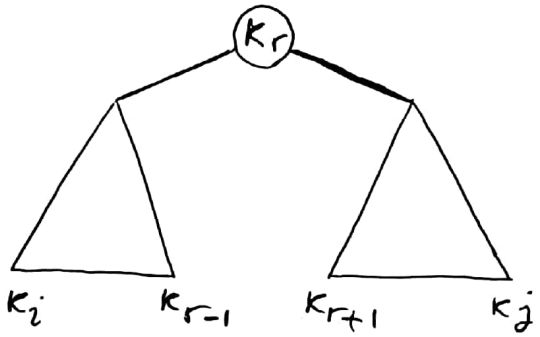
- keyهای $k_i, \dots, k_j$ داده شده است.
- یکی از آنهای k_r، که $i < r \leq j$ به عنوان root قرار می‌گیرد.
- زیردرخت چپ k_r شامل $k_i, \dots, k_{r-1}$ است.
- زیردرخت راست k_r شامل $k_{r+1}, \dots, k_j$ است.



- اگر
  - تمام کاندیداهای ممکن برای root، k_r که $i < r \leq j$ است را بررسی کنیم.
  - تمام BSTهای بهینه که شامل $k_i, \dots, k_{r-1}$ و شامل $k_{r+1}, \dots, k_j$ را تشکیل می‌دهد.

به بوسیله تضمین می‌شود که optimal BST برای $k_i, \dots, k_j$ بدست خواهد آمد.

Recursive solution:

زیرمسئله subproblem:

- optimal BST را برای $k_i, \dots, k_j$ را بیابید که $1 \leq i$، $j \leq n$، $j \geq i-1$
- هنگامی که $j = i-1$ باشد، درخت خالی است.

تعریف می‌کنیم $e[i,j]$ = expected search cost of optimal for $k_i, \ldots, k_j$

اگر $j = i-1$ باشد، آن‌گاه $e[i,j] = 0$

اگر $j \geq i$ باشد،

- یک root، k_r انتخاب می‌کنیم که $i \leq r \leq j$
- یک optimal BST با $k_i, \ldots, k_{r-1}$ به عنوان زیردرخت چپ می‌سازیم.
- یک optimal BST با $k_{r+1}, \ldots, k_j$ به عنوان زیردرخت راست می‌سازیم.

هنگامی که یک زیردرخت، زیردرخت یک node می‌شود:

- عمق هر node در زیردرخت یک واحد بیشتر می‌شود.
- expected search cost افزایش می‌یابد با:

$$w(i,j) = \sum_{l=i}^{j} p_l$$

اگر k_r ریشه optimal BST برای $k_i, \ldots, k_j$ باشد:

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

اما

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

بنابراین:

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

در این رابطه فرض می‌کنیم که می‌دانیم کدام key، k_r است.

ولی نمی دانیم و باید تمام کاندیدیت ها را بررسی کنیم و بهترین را بر داریم

$$e[i,j] = \begin{cases} 0 & \text{if } j = i-1 \\ \min_{i \le r \le j} \{ e[i,r-1] + e[r+1,j] + w(i,j) \} & \text{if } i \le j \end{cases}$$

در این حالت باید یک الگوریتم بازگشتی بنویسیم.

computing an optimal solution :

مثل معمول تمام Value ها را در table ذخیره می کنیم.

$e[\underbrace{1..n+1}_{\text{can store } e[n+1,n]}, \underbrace{0..n}_{\text{can store } e[1,0]}]$

- فقط از مدخل های $e[i,j]$ استفاده خواهد شد، جایی که $i-1 \le j$ باشد.
- همچنین محاسبه می کنیم که

$$root[i,j] = \text{root of subtree with key } k_i, \ldots, k_j$$
$$1 \le i \le j \le n$$

جدول دیگر دوباره $w(i,j)$ را هر بار که نیاز داریم مجدداً محاسبه نمی کند.
(ممکن است $\Theta(j-i)$ اضافه داشته باشیم.)

در عوض :

- Table $w[1..n+1, 0..n]$
- $w[i, i-1] = 0$ for $1 \le i \le n$
- $w[i,j] = w[i,j-1] + p_j$ for $1 \le i \le j \le n$

تمام $\Theta(n^2)$ مقدار در $O(1)$ قابل محاسبه است.

```
OPTIMAL-BST(p, q, n)
for i ← 1 to n+1
    do e[i, i-1] ← 0
       w[i, i-1] ← 0
for l ← 1 to n
    do for i ← 1 to n-l+1
          do j ← i+l-1
             e[i, j] ← ∞
             w[i, j] ← w[i, j-1] + p_j
             for r ← i to j
                do t ← e[i, r-1] + e[r+1, j] + w[i, j]
                   if t < e[i, j]
                      then e[i, j] ← t
                           root[i, j] ← r
return e and root
```

اولین حلقه for مقادیر ابتدائی e و w را برای زیردرخت با 0، key تعریف می‌کند

حلقه for اصلی:

- for را l بار تکرار می‌کند بر روی زیردرختی با l تا key.
- ایده: محاسبه در order سایزهای زیردرخت (کمترین اعداد key و بیشترین n عدد key)

برای مثال در ابتدای شروع.

e	0	1	2	3	4	5
1	0	.25	.65	.8	1.25	2.10
2		0	.2	.3	.75	1.35
3			0	.05	.3	.85
4				0	.2	.7
5					0	.3
6						0

p_i

w	0	1	2	3	4	5
1	0	.25	.45	.5	.7	1.0
2		0	.2	.25	.45	.75
3			0	.05	.25	.55
4				0	.2	.5
5					0	.3
6						0

root	1	2	3	4	5
1	1	1	1	2	2
2		2	2	2	4
3			3	4	5
4				4	5
5					5

Time: $O(n^3)$ حلقه for سه عمق تو در تو است. هر loop کمتر یا مساوی n value in دارد. می‌توان نشان داد که $\Omega(n^3)$ نیز هست. در نتیجه: $\Theta(n^3)$

Construct an optimal solution

```
CONSTRUCT_OPTIMAL_BST(root)
r ← root[1..n]
print "k"_r "is the root"
CONSTRUCT_OPT_SUBTREE(1, r-1, r, "left", root)
CONSTRUCT_OPT_SUBTREE(r+1, n, r, "right", root)
CONSTRUCT_OPT_SUBTREE(i, j, r, dir, root)
if i≤j
    then t ← root[i,j]
         print "k"_t "is" dir "child of k"_r
         CONSTRUCT_OPT_SUBTREE(i, t-1, t, "left", root)
         CONSTRUCT_OPT_SUBTREE(t+1, j, t, "right", root)
```

Elements of dynamic programming

همانطورکه تاکنون بیان شده است

- optimal substructure
- overlaping subproblems