

بسم الله الرحمن الرحيم

دانشگاه صنعتی اصفهان – دانشکده مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۰۱)

طراحی الگوریتم‌ها

حسین فلسفین

حال که دیدیم الگوریتم **brute-force** بسیار ناکارآمد عمل می‌کند، سعی می‌کنیم تا از راهبرد برنامه‌ریزی پویا استفاده کنیم:

Suppose that keys Key_i through Key_j are arranged in a tree that **minimizes**

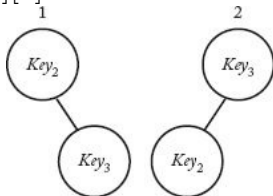
$$\sum_{m=i}^j c_m p_m,$$

where c_m is the number of **comparisons** needed to locate Key_m in the tree. We will call such a tree **optimal for those keys** and denote the optimal value by $A[i][j]$. Because it takes one comparison to locate a key in a tree containing one key, $A[i][i] = p_i$.

Example: Suppose we have three keys. If

$$p_1 = 0.7, \quad p_2 = 0.2, \quad p_3 = 0.1,$$

then, to determine $A[2][3]$ we must consider following two trees:



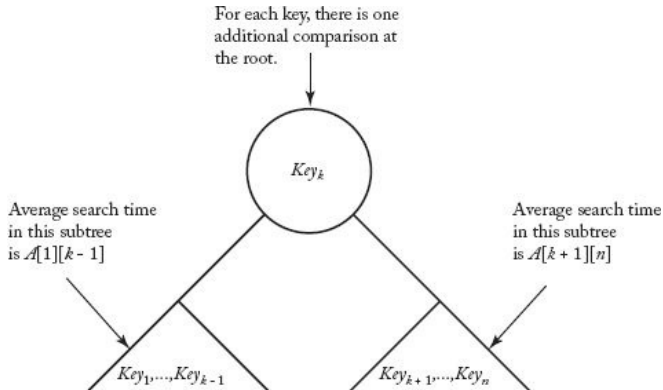
For these two trees we have the following:

1. $1(p_2) + 2(p_3) = 1(0.2) + 2(0.1) = 0.4$
2. $2(p_2) + 1(p_3) = 2(0.2) + 1(0.1) = 0.5$

The first tree is optimal, and $A[2][3] = 0.4$.

Any subtree of an optimal tree must be optimal for the keys in that subtree. 📖 *The principle of optimality applies.*

در درخت بهینه یکی از کلیدها (که نمی‌دانیم کدام است) در ریشه قرار می‌گیرد:



For each $m \neq k$ it takes exactly one more comparison (the one at the root) to location Key_m than it does to locate that key in the subtree that contains it. This one comparison adds $1 \times p_m$ to the average search time for Key_m .

$$\underbrace{A[1][k-1]}_{\text{Average time in left subtree}} + \underbrace{p_1 + \dots + p_{k-1}}_{\text{Additional time comparing at root}} + \underbrace{p_k}_{\text{Average time searching for root}} + \underbrace{A[k+1][n]}_{\text{Average time in right subtree}} + \underbrace{p_{k+1} + \dots + p_n}_{\text{Additional time comparing at root}},$$

which equals

$$A[1][k-1] + A[k+1][n] + \sum_{m=1}^n p_m.$$

The average search time for the optimal tree is given by

$$A[1][n] = \underset{1 \leq k \leq n}{\text{minimum}} (A[1][k-1] + A[k+1][n]) + \sum_{m=1}^n p_m,$$

$A[1][0]$ and $A[n+1][n]$ are defined to be 0. Although the sum of the probabilities in this last expression is clearly 1, we have written it as a sum because we now wish to generalize the result.

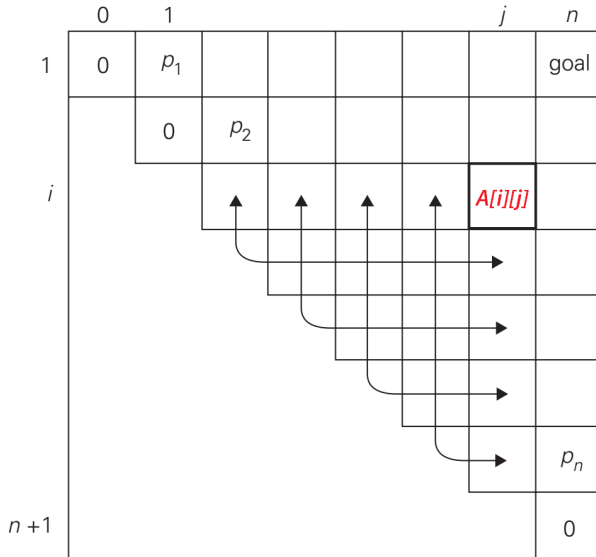
$$A[i][j] = \underset{i \leq k \leq j}{\text{minimum}}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^j p_m \quad i < j$$

$$A[i][i] = p_i$$

$A[i][i-1]$ and $A[j+1][j]$ are defined to be 0.

Because $A[i][j]$ is computed from entries in the i th row but to the left of $A[i][j]$ and from entries in the j th column but beneath $A[i][j]$, we proceed by computing in sequence the values on each diagonal.

عیناً مانند الگوریتمی که برای مسئله ضرب زنجیره‌ای ماتریس‌ها ارائه دادیم.



The array R produced by the algorithm contains the indices of the keys chosen for the root at each step. For example, $R[1][2]$ is the index of the key in the root of an optimal tree containing the first two keys, and $R[2][4]$ is the index of the key in the root of an optimal tree containing the second, third, and fourth keys. *After analyzing the algorithm, we will discuss how to build an optimal tree from R .*

Optimal Binary Search Tree

Problem: Determine an optimal binary search tree for a set of keys, each with a given probability of being the search key.

Inputs: n , the number of keys, and an array of real numbers p indexed from 1 to n , where $p[i]$ is the probability of searching for the i th key.

Outputs: A variable $minavg$, whose value is the average search time for an optimal binary search tree; and a two-dimensional array R from which an optimal tree can be constructed. R has its rows indexed from 1 to $n + 1$ and its columns indexed from 0 to n . $R[i][j]$ is the index of the key in the root of an optimal tree containing the i th through the j th keys.

```

void optsearchtree (int n,
                    const float p[],
                    float& minavg,
                    index R[][])
{
    index i, j, k, diagonal;
    float A[1..n + 1][0..n];

    for (i = 1; i <= n; i++){
        A[i][i - 1] = 0;
        A[i][i] = p[i];
        R[i][i] = i;
        R[i][i - 1] = 0;
    }
    A[n + 1][n] = 0;
    R[n + 1][n] = 0;
    for (diagonal = 1; diagonal <= n - 1; diagonal++){
        for (i = 1; i <= n - diagonal; i++){
            // Diagonal-1 is
            // just above the
            // main diagonal.

            j = i + diagonal;
            
$$A[i][j] = \underset{i \leq k \leq j}{\text{minimum}}(A[i][k - 1] + A[k + 1][j]) + \sum_{m=i}^j p_m.$$

            R[i][j] = a value of k that gave the minimum;
        }
        minavg = A[1][n];
    }
}

```

Every-Case Time Complexity (Optimal Binary Search Tree)**Basic operation:** The instructions executed for each value of k .**Input size:** n , the number of keys.

تحلیل این الگوریتم بسیار شبیه به آنچه برای الگوریتم ضرب زنجیره ای ماتریسها گفتیم است:

$$\sum_{diagonal=1}^{n-1} (n - diagonal)(diagonal + 1) = \frac{n(n-1)(n+4)}{6} \in \Theta(n^3).$$

Recall that R contains the indices of the keys chosen for the root at each step.

Build Optimal Binary Search Tree

Problem: Build an optimal binary search tree.

Inputs: n , the number of keys, an array Key containing the n keys in order, and the array R produced by the previous algorithm. $R[i][j]$ is the index of the key in the root of an optimal tree containing the i th through the j th keys.

Outputs: A pointer tree to an optimal binary search tree containing the n keys.

```
node_pointer tree (index i , j)
{
    index k;
    node_pointer p;

    k = R[i][j];
    if (k == 0)
        return NULL;
    else {
        p = new nodetype;
        p->key = Key[k];
        p->left = tree(i , k - 1);
        p->right = tree(k + 1 , j);
        return p;
    }
}
```

Following our convention for recursive algorithms, the parameters n , Key , and R are not inputs to function `tree`. If the algorithm were implemented by defining n , Key , and R globally, a pointer `root` to the root of an optimal binary search tree is obtained by calling `tree` as follows: $root = tree(1, n);$

Example:

Don	Isabelle	Ralph	Wally
Key[1]	Key[2]	Key[3]	Key[4]

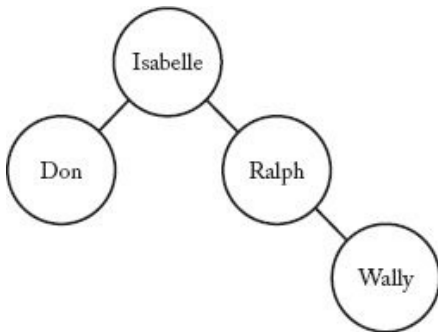
$$p_1 = \frac{3}{8}, \quad p_2 = \frac{3}{8}, \quad p_3 = \frac{1}{8}, \quad p_4 = \frac{1}{8}.$$

	0	1	2	3	4
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1
3			0	$\frac{1}{8}$	$\frac{3}{8}$
4				0	$\frac{1}{8}$
5					0

A

	0	1	2	3	4
1	0	1	1	2	2
2		0	2	2	2
3			0	3	3
4				0	4
5					0

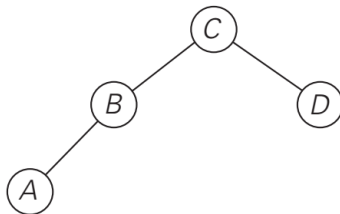
R



Another example:

		key				
		probability				
		A B C D				
		0.1 0.2 0.4 0.3				
		main table				
		0	1	2	3	4
1	0	0.1	0.4	1.1	1.7	
2		0	0.2	0.8	1.4	
3			0	0.4	1.0	
4				0	0.3	
5					0	

		root table				
		0	1	2	3	4
1			1	2	3	3
2				2	3	3
3					3	3
4						4
5						



یک الگوریتم دیگر مبتنی بر راهبرد برنامه‌ریزی پویا برای مسئله کوله‌پشتی ۱ – ۰

The dynamic programming algorithm we designed when we studied the 0-1 Knapsack Problem earlier uses subproblems of the form $F(i, w)$: the subproblem of finding the maximum value of any solution using a subset of the items $1, 2, 3, \dots, i$ and a knapsack of capacity w .

In our novel approach, we define our subproblems as follows. The subproblem is defined by i and a target value V , and $M(i, V)$ is the smallest knapsack capacity so that one can obtain a solution using a subset of items $\{1, 2, \dots, i\}$ with value at least V .

We will have a subproblem for all $i = 0, 1, 2, \dots, n$ and values $V = 0, 1, \dots, \sum_{j=1}^i v_j$.

If v^* denotes $\max_i v_i$, then we see that the largest V can get in a subproblem is $\sum_{j=1}^n v_j \leq nv^*$. Thus, assuming the values are integral, there are at most $O(n^2v^*)$ subproblems. None of these subproblems is precisely the original instance of Knapsack, but if we have the values of all subproblems $M(n, V)$ for $V = 0, 1, \dots, \sum_i v_i$, then the value of the original problem can be obtained easily: it is the largest value V such that $M(n, V) \leq W$.

It is not hard to give a recurrence for solving the subproblem $M(i, V)$. We consider cases depending on whether or not the last item i is included in the optimal solution.

دو حالت داریم: اگر داشته باشیم $V > \sum_{k=1}^{i-1} v_k$ پس قطعاً باید i امین آیتم نیز در مجموعه ما قرار گیرد. در نتیجه داریم

$$M(i, V) = w_i + M(i - 1, \max(V - v_i, 0)).$$

در غیر اینصورت، یعنی اگر $V \leq \sum_{k=1}^{i-1} v_k$ ، آنگاه ممکن است i امین آیتم در مجموعه ما قرار بگیرد یا نگیرد. پس داریم

$$M(i, V) = \min (M(i - 1, V), w_i + M(i - 1, \max(V - v_i, 0)))$$

We can then write down our dynamic programming algorithm. Our algorithm takes $O(n^2 v^*)$ time and correctly computes the optimal values of the subproblems. As was done before, we can trace back through the table M containing the optimal values of the subproblems, to find an optimal solution.

```
Knapsack(n, W, w[], v[]) :  
    Array M[0, 1, ..., n][0, 1, ..., V]  
    For i = 0, 1, ..., n  
        M[i, 0] = 0  
    Endfor  
    For i = 1, 2, ..., n  
        For V = 1, 2, ...,  $\sum_{k=1}^i v_k$   
            If  $V > \sum_{k=1}^{i-1} v_k$  then  
                M[i, V] =  $w_i + M[i - 1, \max(0, V - v_i)]$   
            Else  
                M[i, V] =  $\min(M[i - 1, V], w_i + M[i - 1, \max(0, V - v_i)])$   
            Endif  
        Endfor  
    Endfor  
    Return the maximum value V such that  $M[n, V] \leq W$ 
```

Capacity = 16

item	weight	value
1	2	4
2	5	3
3	10	5
4	5	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0													
1	0	2	2	2	2									
2	0	2	2	2	2	7	7	7						
3	0	2	2	2	2	7	7	7	12	12	17	17	17	
4	0	2	2	2	2	7	7	7	12	12	17	17	17	22

Therefore, the answer is 9.