

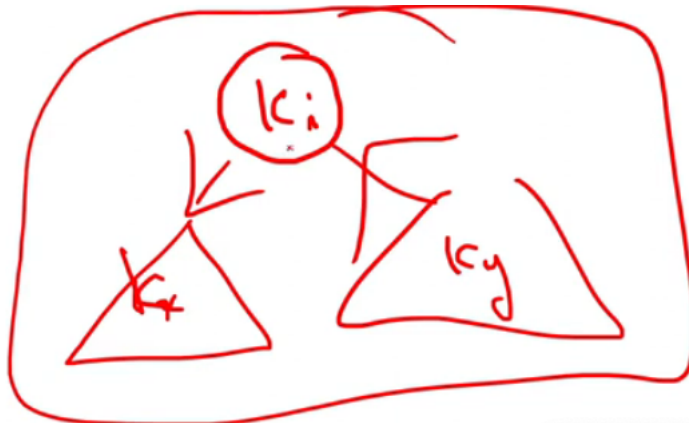
## 챕터 6. Dynamic Programming (동적 계획법) - (2) 최적 이진검색트리

### 최적 이진검색트리 문제

- 주어진  $n$ 개의 키로 최적 이진검색트리를 구하시오.
- 엄밀한 문제 정의
  - 주어진  $n$ 개의 키:  $K_1, K_2, \dots, K_n$
  - 각 키의 검색 확률  $P_i$ : 전체 검색 횟수 중에서  $K_i$ 를 검색하는 확률
  - 각 키의 비교 횟수  $C_i$ :  $K_i$ 를 검색하는 데 필요한 키의 비교 횟수
  - 각 키의 평균 검색 비용(시간): 검색 확률  $\times$  비교 횟수 ( $P_i \times C_i$ )
  - 전체 키의 평균 검색 비용(시간):  $T_{avg} = \sum_{i=1}^n P_i C_i$
- 최적 이진검색트리 문제는 **최적화 문제**
  - 전체 키의 평균 검색 비용을 최소화하는 이진검색트리 찾기

### 이진검색트리 (BST: Binary Search Tree)

- 다음의 조건들을 모두 만족하는 이진트리
  - 각 노드는 하나의 **유일한 키**를 가지고 있다.
  - 모든 노드가 가진 키의 값은 그 노드의 왼쪽 서브트리의 키의 값보다 크다.
  - 모든 노드가 가진 키의 값은 그 노드의 오른쪽 서브트리의 키의 값보다 작다.



## 최적 이진검색트리: 단순무식하게 풀기(Brute-Force Approach)

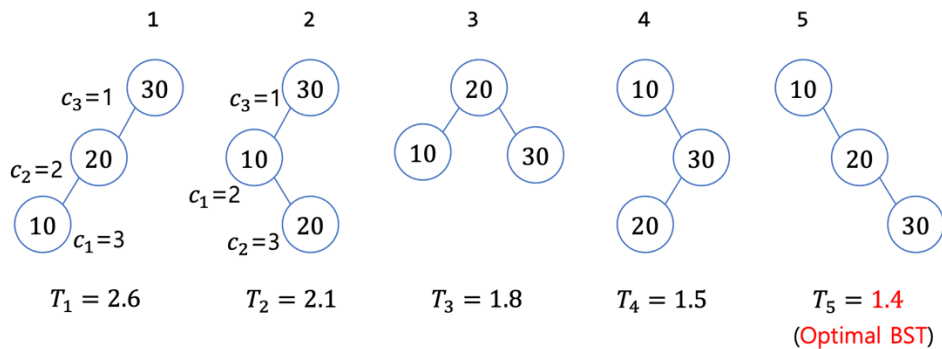
- 모든 경우의 수에 대해서 계산해 보고 최적의 이진트리 선택
- 이진검색트리의 가능한 경우의 수는?

- 카탈란 수:  $C(n) = \frac{1}{n+1} \binom{2n}{n} \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$

- n개의 키로 만들 수 있는 이진 트리의 수 =  $C(n)$

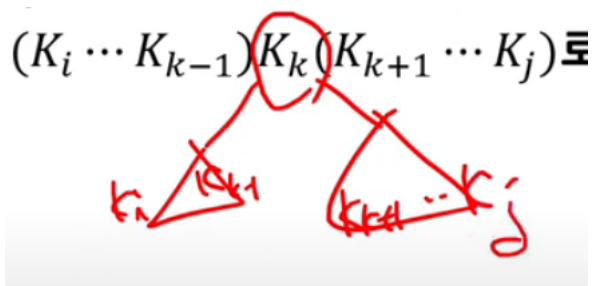
## 최적 이진검색트리의 입력 사례

-  $n = 3$ ,  $K = [10, 20, 30]$ ,  $p = [0.7, 0.2, 0.1]$



## 최적 이진검색트리: 동적계획(Dynamic Programming)

- 1단계: 재귀 관계식을 찾는다.
  - A: 이진검색트리를 만드는데 최적 검색비용의 행렬
  - $A[i][j]$ :  $K_i$ 에서  $K_j$ 까지 이진검색트리를 만드는데 최적 검색 비용
  - 목표:  $K_i \cdots K_j$  ( $K_i \cdots K_{k-1}$ ),  $K_k$ , ( $K_{k+1} \cdots K_j$ )로 분할하는 재귀 관계식 찾기



--> 트리 형태니까 이렇게 재귀식을 짤다.

- 2단계: 상향식 방법으로 해답을 구한다.
  - 초기화:  $A[i][i] = P_i$  (주대각선을  $P_i$ 으로)
  - 최종 목표:  $A[1][n]$ .
  - 상향식 계산: 대각선 1번, 대각선 2번, ---, 대각선  $n - 1$ 번

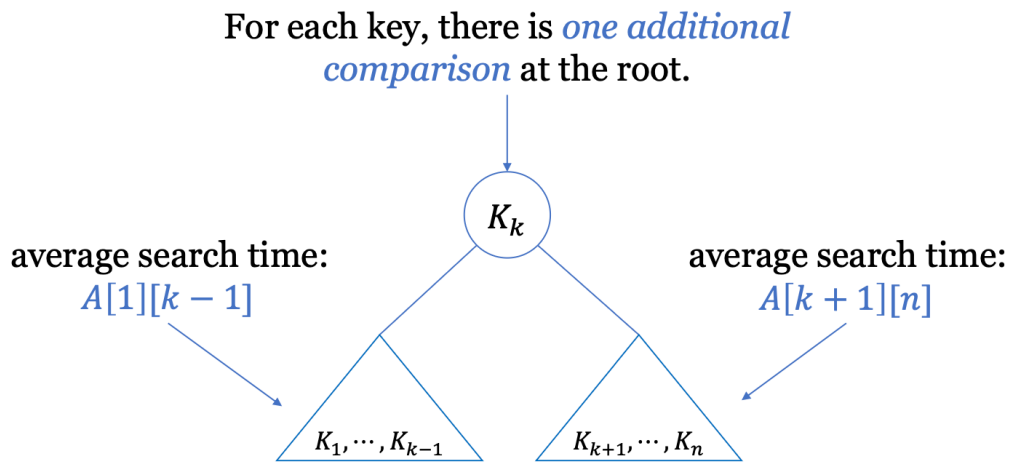
## 최적 이진검색트리의 재귀 관계식 구하기

- 트리 K: 키  $K_k$ 가 루트 노드에 있는 최적 이진검색트리
- 키 비교 횟수: 서브 트리의 비교 횟수에 루트에서 비교 한 번 추가
- $m \neq k$ 인  $K_m$ 에 대해서 트리 k에  $K_m$ 을 놓기 위한 비교 한 번 추가
- $K_m$ 의 평균 검색비용에  $p_m$ 을 추가

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

- 최적 트리: k개의 트리 중 평균 검색비용이 가장 작은 트리

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



### Establish the recursive property:

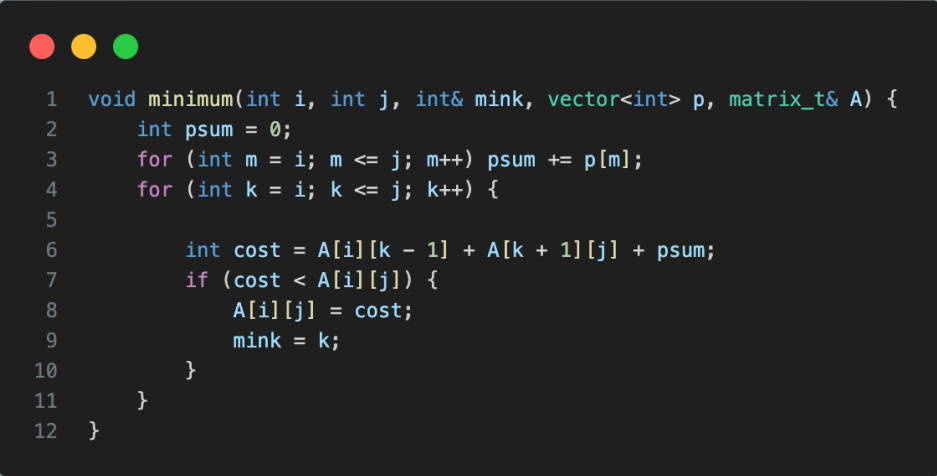
- The average search time for *tree k* is
  - $A[1][k-1] + A[k+1][n] + \sum_{m=1}^n p_m$ .
- The average search time for the *optimal tree* is given by
  - $A[1][n] = \underset{1 \leq k \leq n}{\text{minimum}} (A[1][k-1] + A[k+1][n] + \sum_{m=1}^n p_m)$ ,
    - where  $A[1][0]$  and  $A[n+1][n]$  are defined to be 0.
- In general,
  - $A[i][j] = \underset{i \leq k \leq j}{\text{minimum}} (A[i][k-1] + A[k+1][j]) + \sum_{m=i}^j p_m$ , for  $i < j$ .
  - $A[i][j] = p_i$ ,
  - $A[i][i-1] = A[j+1][j] = 0$ .

```

void optsearchtree(int n, vector<int>& p, matrix_t& A, matrix_t& R)
{
    for (int i = 1; i <= n; i++) {
        A[i][i - 1] = 0; A[i][i] = p[i];
        R[i][i - 1] = 0; R[i][i] = i;
    }
    A[n + 1][n] = R[n + 1][n] = 0;

    for (int diagonal = 1; diagonal <= n - 1; diagonal++)
        for (int i = 1; i <= n - diagonal; i++) {
            int j = i + diagonal;
            A[i][j] = minimum(i, j, k, p, A);
            R[i][j] = k;
        }
}

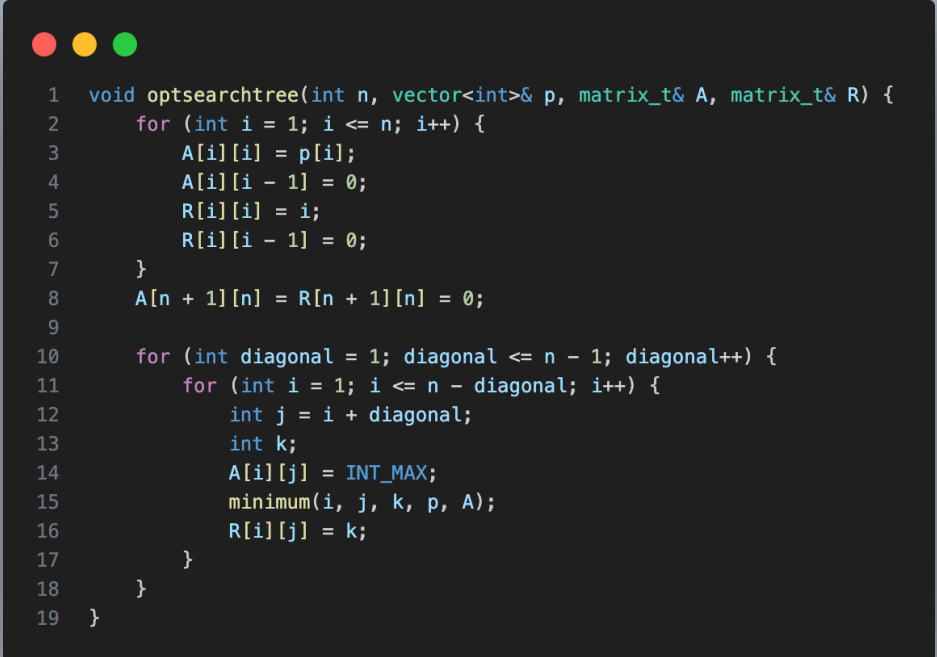
```



```

1 void minimum(int i, int j, int& mink, vector<int> p, matrix_t& A) {
2     int psum = 0;
3     for (int m = i; m <= j; m++) psum += p[m];
4     for (int k = i; k <= j; k++) {
5
6         int cost = A[i][k - 1] + A[k + 1][j] + psum;
7         if (cost < A[i][j]) {
8             A[i][j] = cost;
9             mink = k;
10        }
11    }
12 }

```



```

1 void optsearchtree(int n, vector<int>& p, matrix_t& A, matrix_t& R) {
2     for (int i = 1; i <= n; i++) {
3         A[i][i] = p[i];
4         A[i][i - 1] = 0;
5         R[i][i] = i;
6         R[i][i - 1] = 0;
7     }
8     A[n + 1][n] = R[n + 1][n] = 0;
9
10    for (int diagonal = 1; diagonal <= n - 1; diagonal++) {
11        for (int i = 1; i <= n - diagonal; i++) {
12            int j = i + diagonal;
13            int k;
14            A[i][j] = INT_MAX;
15            minimum(i, j, k, p, A);
16            R[i][j] = k;
17        }
18    }
19 }

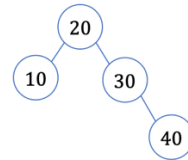
```

A: 최소비용, R: 루트노드, p: 각 키 값의 빈도(frequency)

•  $n=4$ ,  $K = K_1, K_2, K_3, K_4 = [10, 20, 30, 40]$ ,  $p = p_1, p_2, p_3, p_4 = [3, 3, 1, 1]$ .

<i>A</i>	1	2	3	4	<i>R</i>	1	2	3	4
1					1				
2				*	2				
3					3				
4					4				

<b>A</b>	1	2	3	4	<b>R</b>	1	2	3	4
1	3	9	11	14	1	1	1	2	2
2		3	5	8	2		2	2	2
3			1	3	3			3	3
4				1	4				4



preorder: 20 10 30 40  
inorder: 10 20 30 40

Handwritten notes and code for optimal binary search tree construction:

```

INF = 999
keys = [0, 10, 20, 30, 40, 50]
p = [0, 35, 12, 22, 8, 15]
A, R = optsearchtree(p)
print('A = ')
for i in range(1, len(A)):
    print(A[i])
print('R = ')
for i in range(1, len(R)):
    print(R[i])
  
```

Handwritten annotations include red circles around '10', '20', '30', '40' in the keys array, and red arrows pointing to specific elements in the A and R arrays. There are also handwritten labels like 'k1', 'k2', 'k3', 'k4', 'k5' and 'k6'.

Printed output for A and R:

```

A =
[0, 35, 59, 115, 139, 182]
[-1, 0, 12, 46, 62, 100]
[-1, -1, 0, 22, 38, 76]
[-1, -1, -1, 0, 8, 31]
[-1, -1, -1, -1, 0, 15]
[-1, -1, -1, -1, -1, 0]
R =
[0, 1, 1, 1, 1, 3]
[-1, 0, 2, 3, 3, 3]
[-1, -1, 0, 3, 3, 3]
[-1, -1, -1, 0, 4, 5]
[-1, -1, -1, -1, 0, 5]
[-1, -1, -1, -1, -1, 0]
  
```

## 최적 이진검색트리 구하기

- $R[i][j]$ :  $i$ 번째에서  $j$ 번째까지 최적 트리의 루트 노드 인덱스
- 재귀 호출을 통한 **분할 정복**

### ALGORITHM 3.10: Build Optimal Binary Search Tree

```

node_ptr tree(int i, int j, vector<int>& keys, matrix_t& R)
{
    int k = R[i][j];
    if (k == 0)
        return NULL;
    else {
        node_ptr node = create_node(keys[k]);
        node->left = tree(i, k - 1, keys, R);
        node->right = tree(k + 1, j, keys, R);
        return node;
    }
}
  
```



```
1 void preorder(node_ptr root) {
2     if (root == nullptr) return;
3     cout << root->data;
4     if (cnt != N) {
5         cout << " ";
6         cnt++;
7     }
8     preorder(root->left);
9     preorder(root->right);
10 }
11
12 void inorder(node_ptr root) {
13     if (root == nullptr) return;
14     inorder(root->left);
15     cout << root->data;
16     if (cnt != N) {
17         cout << " ";
18         cnt++;
19     }
20     inorder(root->right);
21 }
```

```

1 // Optimal Binary Search Tree
2 /* input case
3 4
4 10 20 30 40
5 3 3 1 1
6 */
7
8 #include <iostream>
9 #include <vector>
10 #include <limits.h>
11 using namespace std;
12
13 #define MAX 999
14 typedef vector<vector<int> > matrix_t;
15 int cnt = 1, N;
16
17 typedef struct node {
18     int data;
19     struct node* left;
20     struct node* right;
21 } *node_ptr;
22
23 node_ptr create_node(int key) {
24     node_ptr node = new struct node;
25     node->data = key;
26     node->left = node->right = nullptr;
27     return node;
28 }
29
30 void minimum(int i, int j, int& mink, vector<int> p, matrix_t& A) {
31     int psum = 0;
32     for (int m = i; m <= j; m++) psum += p[m];
33     for (int k = i; k <= j; k++) {
34         int cost = A[i][k - 1] + A[k + 1][j] + psum;
35         if (cost < A[i][j]) {
36             A[i][j] = cost;
37             mink = k;
38         }
39     }
40 }
41
42 void optsearchtree(int n, vector<int>& p, matrix_t& A, matrix_t& R) {
43     for (int i = 1; i <= n; i++) {
44         A[i][i] = p[i];
45         A[i][i - 1] = 0;
46         R[i][i] = i;
47         R[i][i - 1] = 0;
48     }
49     A[n + 1][n] = R[n + 1][n] = 0;
50
51     for (int diagonal = 1; diagonal <= n - 1; diagonal++) {
52         for (int i = 1; i <= n - diagonal; i++) {
53             int j = i + diagonal;
54             int k;
55             A[i][j] = INT_MAX;
56             minimum(i, j, k, p, A);
57             R[i][j] = k;
58         }
59     }
60 }
61
62 node_ptr tree(int i, int j, vector<int>& keys, matrix_t& R) {
63     int k = R[i][j];
64     if (k == 0) return nullptr;
65
66     node_ptr node = create_node(keys[k]);
67     node->left = tree(i, k - 1, keys, R);
68     node->right = tree(k + 1, j, keys, R);
69     return node;
70 }
71
72 void preorder(node_ptr root) {
73     if (root == nullptr) return;
74     cout << root->data;
75     if (cnt != N) {
76         cout << " ";
77         cnt++;
78     }
79     preorder(root->left);
80     preorder(root->right);
81 }

```



```

1 void inorder(node_ptr root) {
2     if (root == nullptr) return;
3     inorder(root->left);
4     cout << root->data;
5     if (cnt != N) {
6         cout << " ";
7         cnt++;
8     }
9     inorder(root->right);
10 }
11
12 int main() {
13     int n;
14     cin >> n;
15     N = n;
16
17     vector<int> keys(n + 1); // 1-based indexing
18     vector<int> p(n + 1);    // weights or frequencies
19
20     for (int i = 1; i <= n; i++)
21         cin >> keys[i];
22     for (int i = 1; i <= n; i++)
23         cin >> p[i];
24
25     matrix_t A(n + 2, vector<int>(n + 1, 0));
26     matrix_t R(n + 2, vector<int>(n + 1, 0));
27
28     optsearchtree(n, p, A, R);
29     node_ptr root = tree(1, n, keys, R);
30
31     for (int i = 1; i <= n+1; i++) {
32         for (int j = i-1; j <= n; j++) {
33             cout << A[i][j];
34             if (j != n) cout << " ";
35         }
36         cout << endl;
37     }
38     for (int i = 1; i <= n+1; i++) {
39         for (int j = i-1; j <= n; j++) {
40             cout << R[i][j];
41             if (j != n) cout << " ";
42         }
43         cout << endl;
44     }
45     cout << A[1][n] << endl;
46     preorder(root);
47     cout << endl;
48     cnt = 1;
49     inorder(root);
50     cout << endl;
51
52     return 0;
53 }

```