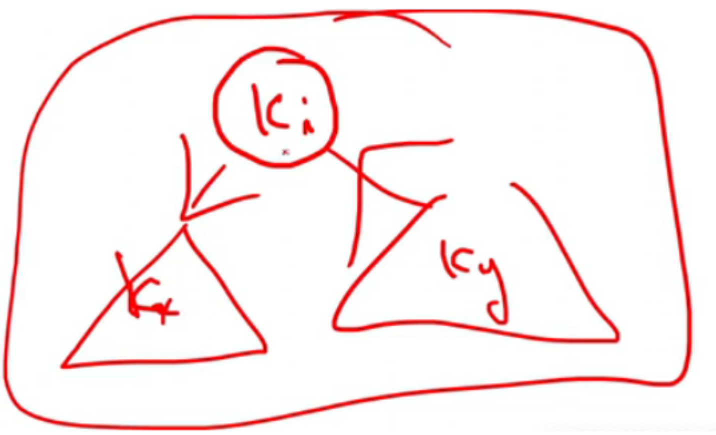
챕터 6. Dynamic Programming (동적 계획법) - (2) 최적 이진검색트리

최적 이진검색트리 문제

- 주어진 n개의 키로 최적 이진검색트리를 구하시오.
- 엄밀한 문제 정의
    - 주어진 n개의 키: K_1, K_2, ---, K_n
    - 각 키의 검색 확률 Pi: 전체 검색 횟수 중에서 Ki를 검색하는 확률
    - 각 키의 비교 횟수 C_i: K_i를 검색하는 데 필요한 키의 비교 횟수
    - 각 키의 평균 검색 비용(시간): 검색 확률 X 비교 횟수 (P_i X C_i)
    - 전체 키의 평균 검색 비용(시간): $T_{avg} = \sum^{n}_{i=1} P_iC_i$
- 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 키를 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.

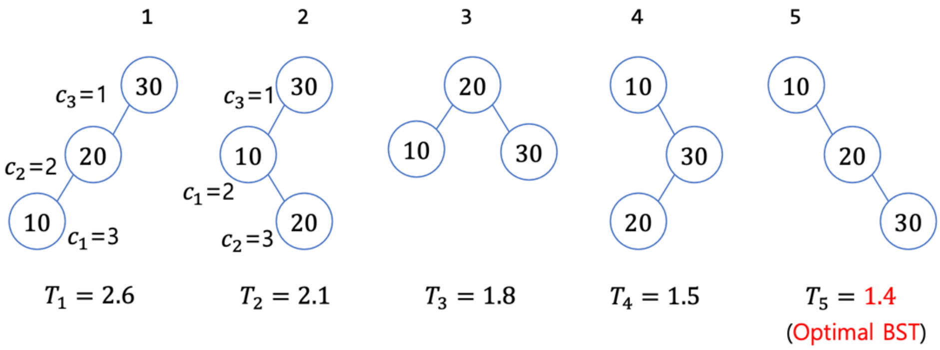


최적 이진검색트리: 단순무식하게 풀기(Brute-Force Approach)

- 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)

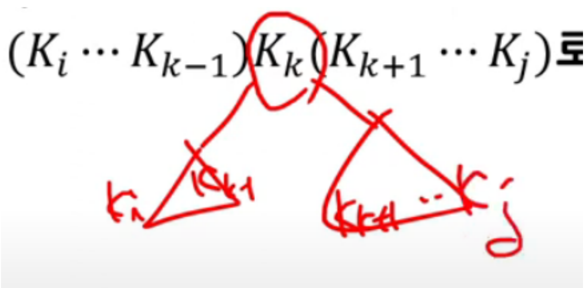
최적 이진검색트리의 입력 사례

- 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]: Ki에서 Kj까지 이진검색트리를 만드는데 최적 검색 비용
    - 목표: K_i --- K_j를 (K_i --- K_{k-1})K_k(K_{k+1} --- K_j)로 분할하는 재귀 관계식 찾기



--> 트리 형태니까 이렇게 재귀식을 짠다.

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

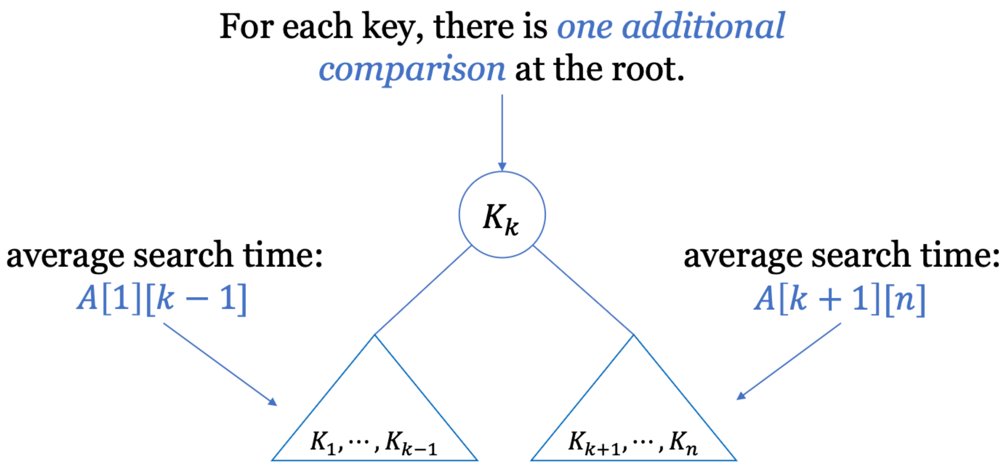
최적 이진검색트리의 재귀 관계식 구하기

- 트리 K: 키 Kk가 루트 노드에 있는 최적 이진검색트리
- 키 비교 횟수: 서브 트리의 비교 횟수에 루트에서 비교 한 번 추가
- m != k인 Km에 대해서 트리 k에 Km을 놓기 위한 비교 한 번 추가
  - K_m의 평균 검색비용에 P_m을 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

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



- **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
    - $A[1][n] = \underset{1\le k\le n}{\text{minimum}}(A[1][k-1]+A[k+1][n])+\sum_{m=1}^{n} p_m$,
      - where $A[1][0]$ and $A[n+1][n]$ are defined to be 0.
  - In general,
    - $A[i][j] = \underset{i\le k\le j}{minimum}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m$, 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;
        }
}
```

```
void minimum(int i, int j, int& mink, vector<int> p, matrix_t& A) {
    int psum = 0;
    for (int m = i; m <= j; m++) psum += p[m];
    for (int k = i; k <= j; k++) {

        int cost = A[i][k - 1] + A[k + 1][j] + psum;
        if (cost < A[i][j]) {
            A[i][j] = cost;
            mink = k;
        }
    }
}
```

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

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

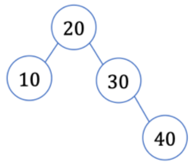
- $n = 4$, K $= K_1, K_2, K_3, K_4$=[10, 20, 30, 40], $p = p_1, p_2, p_3, p_4$=[3, 3, 1, 1].

A	1	2	3	4
1				
2				
3				
4				

R	1	2	3	4
1				
2				
3				
4				

A	1	2	3	4
1	3	9	11	14
2		3	5	8
3			1	3
4				1

R	1	2	3	4
1	1	1	2	2
2		2	2	2
3			3	3
4				4




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

```
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])
```

```
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]: i번째에서 j번째까지 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;
    }
}
```

```cpp
void preorder(node_ptr root) {
    if (root == nullptr) return;
    cout << root->data;
    if (cnt != N) {
        cout << " ";
        cnt++;
    }
    preorder(root->left);
    preorder(root->right);
}

void inorder(node_ptr root) {
    if (root == nullptr) return;
    inorder(root->left);
    cout << root->data;
    if (cnt != N) {
        cout << " ";
        cnt++;
    }
    inorder(root->right);
}
```

```cpp
// Optimal Binary Search Tree
/* input case
4
10 20 30 40
3 3 1 1
*/

#include <iostream>
#include <vector>
#include <limits.h>
using namespace std;

#define MAX 999
typedef vector<vector<int> > matrix_t;
int cnt = 1, N;

typedef struct node {
    int data;
    struct node* left;
    struct node* right;
} *node_ptr;

node_ptr create_node(int key) {
    node_ptr node = new struct node;
    node->data = key;
    node->left = node->right = nullptr;
    return node;
}

void minimum(int i, int j, int& mink, vector<int> p, matrix_t& A) {
    int psum = 0;
    for (int m = i; m <= j; m++) psum += p[m];
    for (int k = i; k <= j; k++) {
        int cost = A[i][k - 1] + A[k + 1][j] + psum;
        if (cost < A[i][j]) {
            A[i][j] = cost;
            mink = k;
        }
    }
}

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

node_ptr tree(int i, int j, vector<int>& keys, matrix_t& R) {
    int k = R[i][j];
    if (k == 0) return nullptr;

    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;
}

void preorder(node_ptr root) {
    if (root == nullptr) return;
    cout << root->data;
    if (cnt != N) {
        cout << " ";
        cnt++;
    }
    preorder(root->left);
    preorder(root->right);
}
```

```
void inorder(node_ptr root) {
    if (root == nullptr) return;
    inorder(root->left);
    cout << root->data;
    if (cnt != N) {
        cout << " ";
        cnt++;
    }
    inorder(root->right);
}

int main() {
    int n;
    cin >> n;
    N = n;

    vector<int> keys(n + 1);  // 1-based indexing
    vector<int> p(n + 1);     // weights or frequencies

    for (int i = 1; i <= n; i++)
        cin >> keys[i];
    for (int i = 1; i <= n; i++)
        cin >> p[i];

    matrix_t A(n + 2, vector<int>(n + 1, 0));
    matrix_t R(n + 2, vector<int>(n + 1, 0));

    optsearchtree(n, p, A, R);
    node_ptr root = tree(1, n, keys, R);

    for (int i = 1; i <= n+1; i++) {
        for (int j = i-1; j <= n; j++) {
            cout << A[i][j];
            if (j != n) cout << " ";
        }
        cout << endl;
    }
    for (int i = 1; i <= n+1; i++) {
        for (int j = i-1; j <= n; j++) {
            cout << R[i][j];
            if (j != n) cout << " ";
        }
        cout << endl;
    }
    cout << A[1][n] << endl;
    preorder(root);
    cout << endl;
    cnt = 1;
    inorder(root);
    cout << endl;

    return 0;
}
```