Chapter 3. (Part 2)

Dynamic Programming

경북대학교 배준현 교수

(joonion@knu.ac.kr)

Contents

카탈랑 수: *Catalan number*

경우의 수를 구하는 여러 조합 문제에서 자주 등장하는 카탈랑 수를 통해 비슷한 DP를 연습해보자.

카탈랑 수가 등장하는 조합 문제:

- 괄호 문제: 괄호를 올바르게 짝을 맞추는 경우의 수 문제
- BST 문제: 이진 검색 트리의 경우의 수 문제
- 스택 순열 문제: 스택을 통과하여 만들 수 있는 순열의 경우의 수 문제
- 연쇄 행렬 곱셈 문제: 행렬의 곱셈 순서를 만들 수 있는 경우의 수 문제
- 삼각형 분할 문제: $n + 2$개의 정점으로 이루어진 다각형을 삼각형으로 분할하는 방법의 수

Dynamic Programming

$$C_0 = 1$$

$$C_n = \sum_{i=0}^{n-1} C_i \times C_{n-1-i}\,, n \geq 1$$

카탈랑 수 C_n을 1,000,000,007로 나눈 값을 구하시오.

1. 재귀로 풀기
2. 메모이제이션 적용하기
3. 태뷸레이션 적용하기
4. 일반항 찾아보기

$C_0 = 1$

$C_1 = C_0 \times C_0 = 1$

$C_2 = C_0 \times C_1 + C_1 \times C_0 = 2$

$C_3 = C_0 \times C_2 + C_1 \times C_1 + C_2 \times C_0 = 5$

$C_4 = C_0 \times C_3 + C_1 \times C_2 + C_2 \times C_1 + C_3 \times C_0 = 14$

$C_5 = 42$

…

$C_{10} = 16{,}796$

…

$C_{20} = 6{,}564{,}120{,}420$

…

$C_{50} = 1{,}978{,}261{,}657{,}756{,}160{,}653{,}623{,}774{,}456$

…

3.4 Chained Matrix Multiplication

- The *Multiplication* of *Chained Matrices*:
  - Suppose we want to multiply a 2×3 matrix and a 3×4 matrix:
    - $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$
  - The resultant matrix is a 2×4 matrix.
  - If we use the standard method of multiplying matrices
    - it takes *three* elementary *multiplications*.
    - For 29 in the product: 3 multiplications $(1 \times 7 + 2 \times 2 + 3 \times 6)$.
  - Because there $2 \times 4 = 8$ entries in the product,
    - the *total number* of elementary multiplication is $2 \times 3 \times 4 = 24$.

3.4 Chained Matrix Multiplication

- The number of *elementary multiplications*:
  - In general, to multiply an $i \times j$ matrix with a $j \times k$ matrix,
    - the number of elementary multiplications is $i \times j \times k$.
  - Consider the multiplication of the following four matrices:
    - $A \quad \times \quad B \quad \times \quad C \quad \times \quad D$
    - $(20 \times 2) \quad (2 \times 30) \quad (30 \times 12) \quad (12 \times 8)$
  - Matrix multiplication is an *associative operation*,
    - meaning the *order* in which we multiply *does not matter*.
    - for example, $A(B(CD)) = (AB)(CD)$.

3.4 Chained Matrix Multiplication

- The number of elementary multiplications:
  - There are *five different orders* with different number multiplications.
    - $A\big(B(CD)\big)$: 3,680
    - $(AB)(CD)$: 8,880
    - $A\big((BC)D\big)$: 1,232 *(optimal order)*
    - $\big((AB)C\big)D$: 10,320
    - $\big(A(BC)\big)D$: 3,120
  - *Our goal* is to develop an algorithm that
    - determines the *optimal order* for multiplying n chained matrices.

3.4 Chained Matrix Multiplication

- The *Brute-Force* Approach:
  - Consider *all possible orders* and *take* the *minimum*.
  - We will show that this algorithm is *at least exponential-time*.
    - For a given n matrices: $A_1, A_2, \cdots, A_n$,
    - let t_n be the number of different orders.
    - Then, $t_n \geq t_{n-1} + t_{n-1} = 2t_{n-1}$, and $t_2 = 1$.
    - Therefore, $t_n \geq 2^{n-2}$. (*at least exponential*)

3.4 Chained Matrix Multiplication

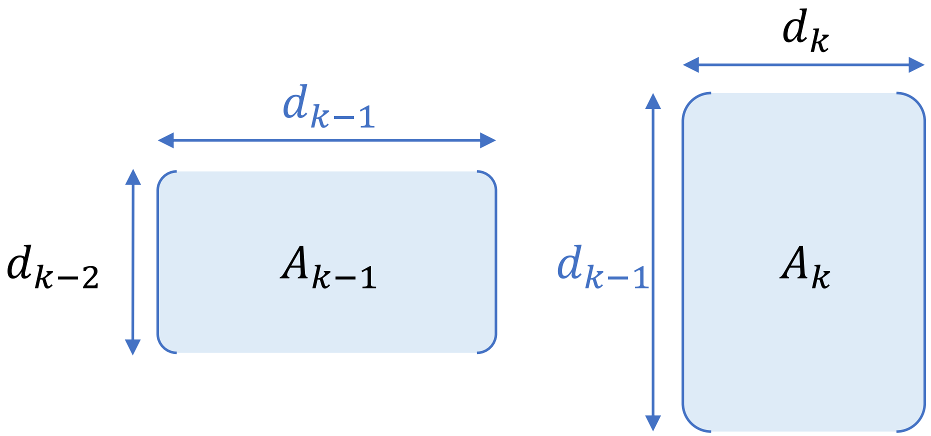
- Does the *principle of optimality* apply?
  - Show that the *optimal order* for multiplying n matrices includes
    - the *optimal order* for multiplying any *subset* of the n matrices.
  - Then, we can use dynamic programming to construct a solution.

$$A_1\left(\left(\boxed{\left((A_2A_3)A_4\right)}A_5\right)A_6\right)$$

- If this order is *optimal* for multiplying $A_1A_2A_3A_4A_5A_6$,
- then the *optimal order* for multiplying $A_2A_3A_4$ is this.

$$\boxed{\left((A_2A_3)A_4\right)}$$

- Multiplying A_{k-1} times A_k:
  - The number of columns in A_{k-1}
    - must equal to the number of rows in A_k.
  - If we let d_0 be the number of rows in A_1
    - and d_k be the number of columns in A_k for $1 \le k \le n$,
    - then, the dimension of A_k is $d_{k-1} \times d_k$.



3.4 Chained Matrix Multiplication

- Creating a sequence of arrays:
  - For $1 \le i \le j \le n$, let

$$M[i][j] = \begin{cases} \textit{minimum number} \text{ of } \textit{multiplications} \text{ needed to multiply} \\ A_i \text{ through } A_j, \text{ if } i < j. \\ 0, \text{ if } i = j. \end{cases}$$

$$\begin{array}{ccccccccccc} A_1 & \times & A_2 & \times & A_3 & \times & A_4 & \times & A_5 & \times & A_6 \\ (5 \times 2) & & (2 \times 3) & & (3 \times 4) & & (4 \times 6) & & (6 \times 7) & & (7 \times 8) \\ d_0 \quad d_1 & & d_2 & & d_3 & & d_4 & & d_5 & & d_6 \end{array}$$

- To multiply A_4, A_5, and A_6, we have two orders:
  - $(A_4A_5)A_6$: $d_3d_4d_5 + d_3d_5d_6 = 392$
  - $A_4(A_5A_6)$: $d_4d_5d_6 + d_3d_4d_6 = 528$
- Therefore, $M[4][6] = minimum(392, 528) = 392$.

3.4 Chained Matrix Multiplication

- Establish the recursive property:
  - The *optimal order* for multiplying *six* matrices
    - must have *one of these factorizations*:

1. $(A_1)(A_2A_3A_4A_5A_6)$
2. $(A_1A_2)(A_3A_4A_5A_6)$
3. $(A_1A_2A_3)(A_4A_5A_6)$
4. $(A_1A_2A_3A_4)(A_5A_6)$
5. $(A_1A_2A_3A_4A_5)(A_6)$

  - Of these *factorizations*,
    - the one that *yields* the *minimum* number of multiplications
      - must be the optimal one.

3.4 Chained Matrix Multiplication

- Establish the *recursive property*:
  - The number of multiplications for the kth factorization
    - is the *minimum number* needed to *obtain each factor*
      - ***plus*** the *number* needed to *multiply two factors*.
    - that is, $M[1][k] + M[k+1][6] + d_0 d_k d_6$.
  - We have established that
    - $M[1][6] = \underset{1 \le k \le 5}{\text{minimum}}(M[1][k] + M[k+1][6] + d_0 d_k d_6)$.
  - We can *generalize* this result into:
    - $M[i][j] = \underset{i \le k \le j-1}{\text{minimum}}\left(M[i][k] + M[k+1][j] + d_{i-1} d_k d_j\right)$, if $i < j$.
    - $M[i][i] = 0$.

3.4 Chained Matrix Multiplication

M	1	2	3	4	5	6	
1	0	30	64	132	226	**348**	*diagonal* 5
2		0	24	72	156	268	*diagonal* 4
3			0	72	198	366	*diagonal* 3
4				0	168	392	*diagonal* 2
5					0	336	*diagonal* 1
6						0	*main diagonal*

3.4 Chained Matrix Multiplication

$M[i][i] = 0$ for $1 \leq i \leq 6$

$$M[1][2] = \underset{1 \leq k \leq 1}{\text{minimum}}(M[1][k] + M[k+1][2] + d_0 d_k d_2)$$
$$= M[1][1] + M[2][2] + d_0 d_1 d_2 = 0 + 0 + 5 \times 2 \times 3 = 30.$$

$$M[1][3] = \underset{1 \leq k \leq 2}{\text{minimum}}(M[1][k] + M[k+1][3] + d_0 d_k d_3)$$
$$= \text{minimum}(M[1][1] + M[2][3] + d_0 d_1 d_3, M[1][2] + M[3][3] + d_0 d_2 d_3)$$
$$= \text{minimum}(0 + 24 + 5 \times 2 \times 4,\ 30 + 0 + 5 \times 3 \times 4) = 64.$$

$$M[1][4] = \underset{1 \leq k \leq 3}{\text{minimum}}(M[1][k] + M[k+1][4] + d_0 d_k d_4)$$
$$= \text{minimum}(M[1][1] + M[2][4] + d_0 d_1 d_4, M[1][2] + M[3][4] + d_0 d_2 d_4, M[1][3] + M[4][4] + d_0 d_3 d_4)$$
$$= \text{minimum}(0 + 72 + 5 \times 2 \times 6,\ 30 + 72 + 5 \times 3 \times 6,\ 64 + 0 + 5 \times 4 \times 6) = 132.$$

$M[1][6] = 348.$

3.4 Chained Matrix Multiplication

ALGORITHM 3.6: Minimum Multiplications

```
int minmult(int n, int d[], int P[][], int M[][]) {
    int i, j, k, diagonal;

    for (i = 1; i <= n; i++)
        M[i][i] = 0;
    for (diagonal = 1; diagonal <= n - 1; diagonal++)
        for (i = 1; i <= n - diagonal; i++) {
            j = i + diagonal;
            M[i][j] = minimum(M[i][k] + M[k + 1][j] + d_{i-1} d_k d_j);
                     i≤k≤j−1
            P[i][j] = a value of k that gave the minimum;
        }
    return M[1][n];
}
```

$$M[i][j] = \underset{i \le k \le j-1}{\text{minimum}}\left(M[i][k] + M[k+1][j] + d_{i-1} d_k d_j\right)$$

3.4 Chained Matrix Multiplication

```
void minmult(int n, vector<int>& d, matrix_t& M, matrix_t& P)
{
    for (int i = 1; i <= n; i++)
        M[i][i] = 0;
    for (int diagonal = 1; diagonal <= n - 1; diagonal++)
        for (int i = 1; i <= n - diagonal; i++) {
            int j = i + diagonal, k;
            M[i][j] = minimum(i, j, k, d, M);
            P[i][j] = k;
        }
}
```

```
int minimum(int i, int j, int& mink, vector<int>& d, matrix_t& M)
{
    int minValue = INF, value;
    for (int k = i; k <= j - 1; k++) {
        value = M[i][k] + M[k + 1][j] + d[i - 1] * d[k] * d[j];
        if (minValue > value) {
            minValue = value;
            mink = k;
        }
    }
    return minValue;
}
```

3.4 Chained Matrix Multiplication

- Time Complexity of Algorithm 3.6 (Every-Case)
  - Basic Operation: the *instructions* executed for each value of k.
  - Input Size: n, the number of matrices to be multiplied.
  - Since we have a nested loop,
    - the number of passes through the k loop is
      - $j - 1 - i + 1 = i + diagonal - 1 - i + 1 = diagonal.$
    - the total number of times that the basic operation is done equals

$$\sum_{diagonal=1}^{n-1} [(n - diagonal) \times diagonal] = \frac{n(n-1)(n+1)}{6} \in \Theta(n^3)$$

3.4 Chained Matrix Multiplication

- Obtaining the *Optimal Order* from the array P.

P	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5
6						

- P[1][6] = 1: P[1][1] & <u>P[2][6]</u> • $(A_1 A_2 A_3 A_4 A_5 A_6)$
- P[2][6] = 5: <u>P[2][5]</u> & P[6][6] • $(A_1)(A_2 A_3 A_4 A_5 A_6)$
- P[2][5] = 4: <u>P[2][4]</u> & P[5][5] • $(A_1)\big((A_2 A_3 A_4 A_5) A_6\big)$
- P[2][4] = 3: <u>P[2][3]</u> & P[4][4] • $(A_1)\Big(\big((A_2 A_3 A_4) A_5\big) A_6\Big)$
- P[2][3] = 2: P[2][2] & P[3][3] • $(A_1)\Big(\big(((A_2 A_3) A_4) A_5\big) A_6\Big)$

ALGORITHM 3.7: Print Optimal Order

```
void order(int i, int j, matrix_t& P, string& s)
{
    if (i == j)
        s += string("A") + to_string(i);
    else {
        int k = P[i][j];
        s += string("(");
        order(i, k, P, s);
        order(k + 1, j, P, s);
        s += string(")");
    }
}
```

3.5 Optimal Binary Search Trees

- A ***binary search tree*** (**BST**) is
  - a *binary tree* of items (*keys*) that come from an *ordered* set, such that

1. Each node contains *one unique key*.
2. The keys in the *left subtree* of a given node
   - are *less than* or *equal to* the key in that node.
3. The keys in the *right subtree* of a given node
   - are *greater than* or *equal to* the key in that node.

3.5 Optimal Binary Search Trees

- ***Optimal*** Binary Search Tree
  - Our goal is to organize the keys in a BST so that
    - the *average time* it takes to locate a key is *minimized*.
  - Optimal BST is a tree that is organized in this fashion.
  - This problem is *not hard*
    - if all keys have the *same probability* of being the *search key*.
  - We are concerned with the case
    - where the keys *do **not** have the same probability*.
  - We will discuss the case in which
    - it is known that the *search key is **in** the tree*.

ALGORITHM 3.8: Search Binary Tree

```
void search(node_ptr tree, int keyin, node_ptr& p)
{
    bool found;

    p = tree;
    found = false;
    while (!found) {
        if (p->key == keyin)
            found = true;
        else if (keyin < p->key)
            p = p->left;
        else //  keyin > p->key
            p = p->right;
    }
}
```

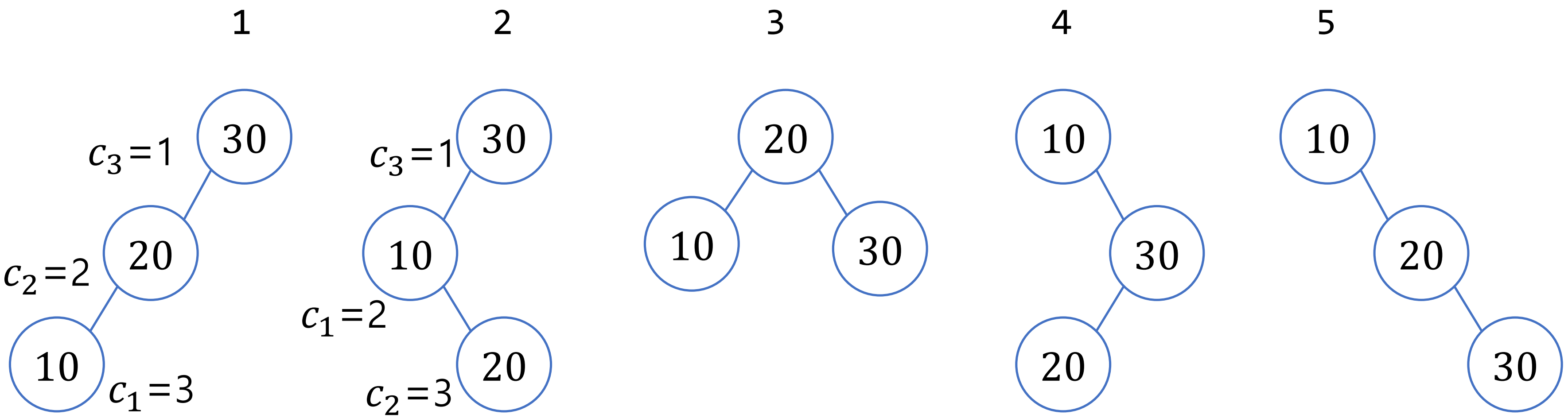
```
typedef struct node *node_ptr;
typedef struct node {
    int key;
    node_ptr left;
    node_ptr right;
} node_t;
```

3.5 Optimal Binary Search Trees

- Evaluating the search time of a BST
  - The *search time* is the *number of comparisons*
    - done by procedure *search* to locate a key.
  - Our goal is to determine a binary search tree
    - for which the *average search time* is *minimal*.
  - Let $K_1, K_2, \cdots, K_n$ be the n keys in order,
    - and let p_i be the *probability* that K_i is the *search key*.
  - The ***average search time*** is $T_{avg} = \sum_{i=1}^{n} c_i\, p_i$
    - where c_i be the *number of comparisons* needed to find K_i in a given tree.

3.5 Optimal Binary Search Trees

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



$T_1 = 2.6$ $T_2 = 2.1$ $T_3 = 1.8$ $T_4 = 1.5$ $T_5 = 1.4$

(Optimal BST)

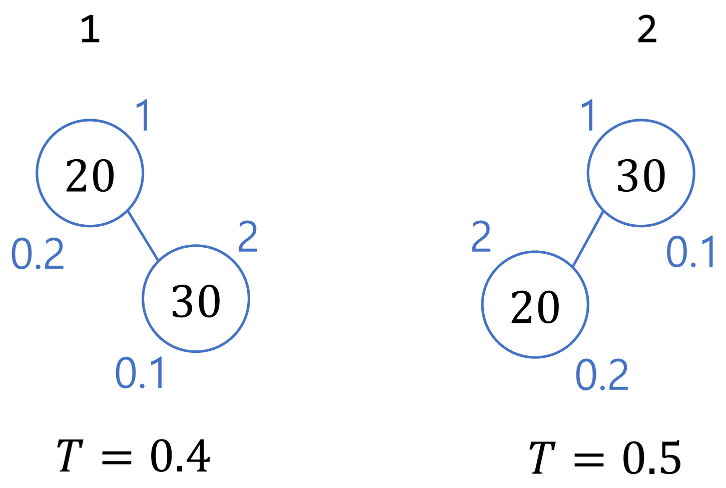
3.5 Optimal Binary Search Trees

- Finding an Optimal BST
  - by considering all binary search trees.
  - The number of *different* BSTs with a depth of $n - 1$ is 2^{n-1}.
    - If a BST's depth is $n - 1$,
      - a node can be either to the left or to the right of its parent.
    - It means there are *two possibilities at each of those levels*.
  - The time complexity of this brute-force approach is *exponential*.

3.5 Optimal Binary Search Trees

- To apply the Dynamic Programming
  - Suppose that keys K_i through K_j are arranged in a tree
    - that minimizes $\sum_{m=i}^{j} c_m p_m$.
  - 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 containing one key,
    - $A[i][i] = p_i$.

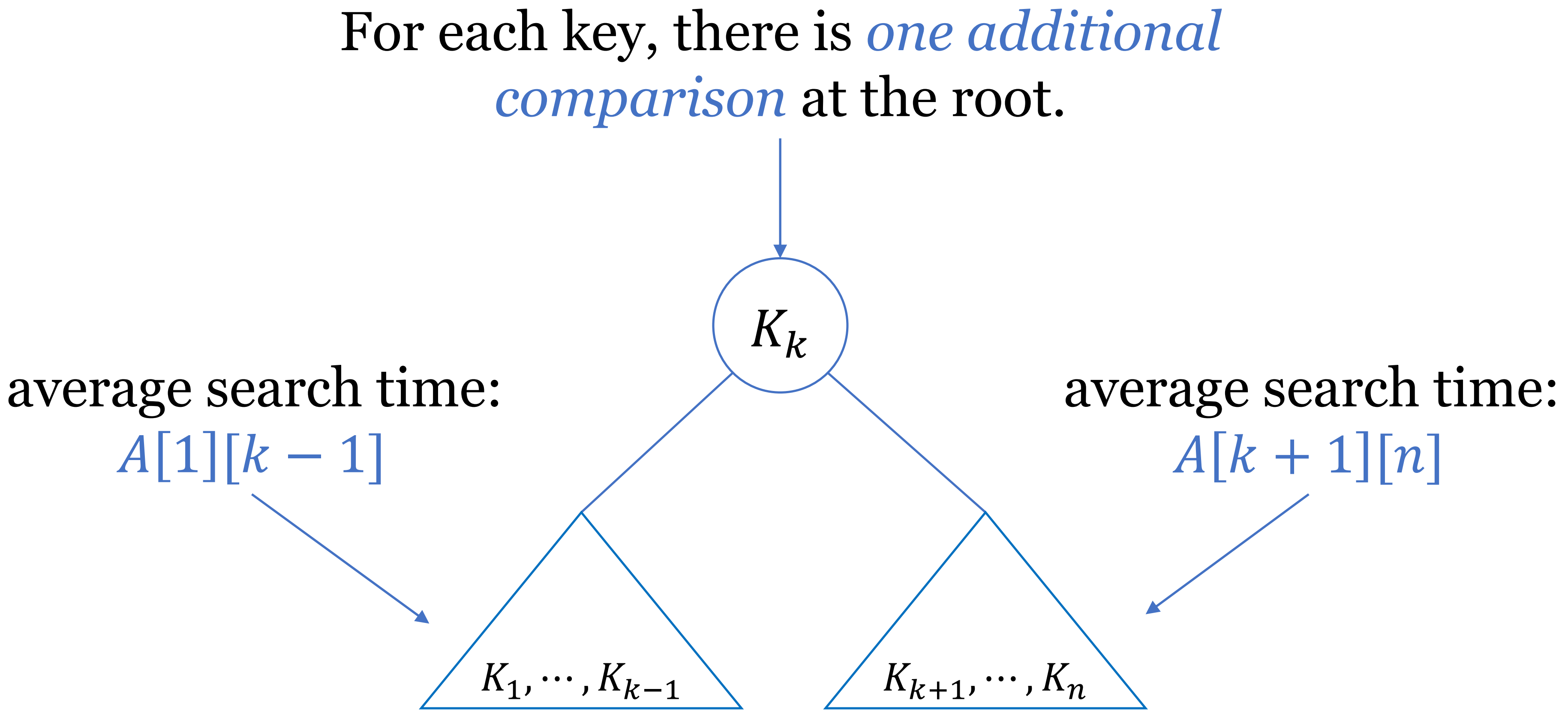
- An example:
  - $n = 3$, K=[10, 20, 30], $p = [0.7, 0.2, 0.1]$, then determine $A[2][3]$.



- $A[2][3] = 0.4$
  1. $1 \times p_2 + 2 \times p_3 = 0.4$ *(optimal)*
  2. $2 \times p_2 + 1 \times p_3 = 0.5$

- Note that the *optimal tree* is
  - the *right subtree* of the optimal tree obtained from the previous tree.
- The *principle of optimality* applies:
  - Any subtree of an optimal tree must be *optimal* for the key in that subtree.

3.5 Optimal Binary Search Trees



3.5 Optimal Binary Search Trees

- 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 $i < j$.
    - $A[i][j] = p_i$,
    - $A[i][i-1] = A[j+1][j] = 0$.

3.5 Optimal Binary Search Trees

- Determining an *Optimal* BST:
  - We proceed by computing *in sequence* the values on *each diagonal*.
    - because $A[i][j]$ is computed
      - from entries in the ith row but to the left of $A[i][j]$,
      - and from entries in the jth column but beneath $A[i][j]$.
  - Let an array R contain
    - the indices of the keys chosen for the root at each step.
    - $R[i][j]$: the index of the key in the root of an optimal tree
      - containing the ith through the jth keys.

3.5 Optimal Binary Search Trees

ALGORITHM 3.9: Optimal Binary Search Tree

```
void optsearchtree(int n, float p[], float &minavg, int R[][MAX]) {
    int i, j, k, diagonal;
    float A[MAX][MAX];
    for (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] = 0;
    R[n + 1][n] = 0;
    for (diagonal = 1; diagonal <= n - 1; diagonal++)
        for (i = 1; i <= n - diagonal; i++) {
            j = i + diagonal;
            A[i][j] = minimum_{i≤k≤j}(A[i][k − 1] + A[k + 1][j]) + ∑_{m=i}^{j} p_m;
            R[i][j] = a value of k that gave the minimum;
        }
    minavg = A[1][n];
}
```

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

3.5 Optimal Binary Search Trees

```
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;
        }
}
```

3.5 Optimal Binary Search Trees

ALGORITHM 3.10: Build Optimal Binary Search Tree

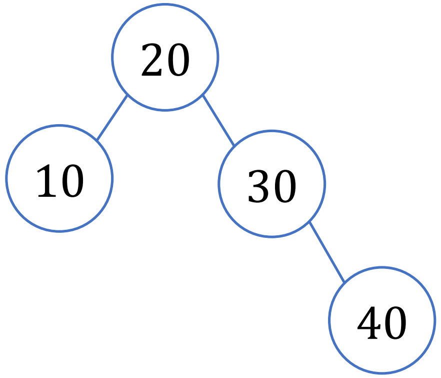
```
node_ptr tree(int i, int j, vector<int>& keys, matrix_t& R)
{
    int k = R[i][j];
    if (k == 0)
        return NULL;
    else {
        node_ptr node = create_node(keys[k]);
        node->left = tree(i, k - 1, keys, R);
        node->right = tree(k + 1, j, keys, R);
        return node;
    }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

3.5 Optimal Binary Search Trees

- An example:
  - $n = 4, K = [10, 20, 30, 40], p = [3, 3, 1, 1]$.

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
```

3.5 Optimal Binary Search Trees

- Time Complexity of Algorithm 3.9:
  - Basic Operation: the instructions executed for *each value of* k.
    - They include a comparison to test for the minimum.
  - Input Size: n, the *number of keys*.
  - The control of this algorithm is almost *identical* to Algorithm 3.6.
    - The only difference is that,
      - the basic operation is done $diagonal + 1$ times.
  - Therefore,
    - $T(n) = \frac{n(n-1)(n+4)}{6} \in \Theta(n^3)$

Any Questions?