개요

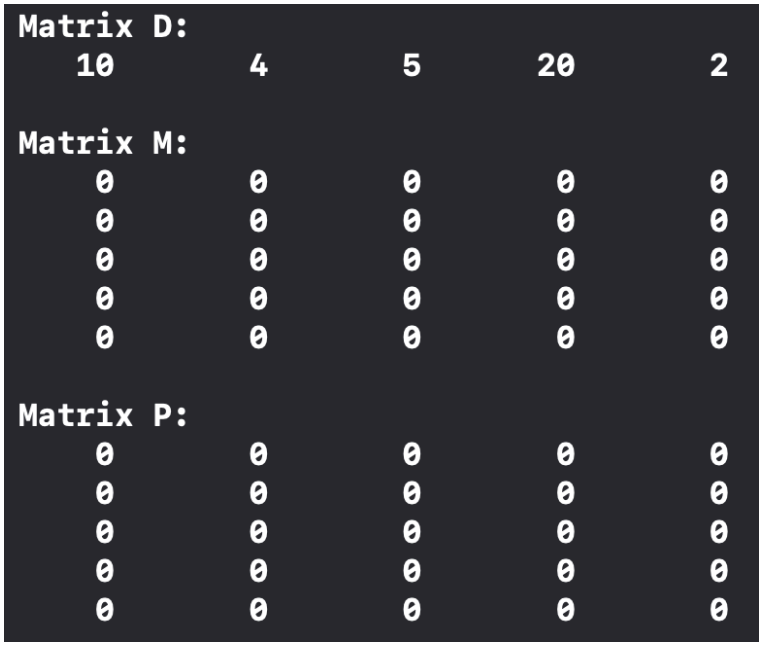
수학과 컴퓨터공학, 그리고 경제학에서 동적 계획법(Dynamic Programming)이란 복잡한 문제를 간단한 여러개의 문제로 나누어 푸는 방법을 말한다. 이것은 부분 문제 반복과 최적 부분 구조를 가지고 있는 알고리즘을 일반적인 방법에 비해 더욱 적은 시간 내에 풀 때 사용한다. 동적 계획법의 원리는 매우 간단하다. 일반적으로 주어진 문제를 풀기 위해서, 문제를 여러 개의 하위 문제로 나누어 푼 다음, 그것을 결합하여 최종적인 목적에 도달하는 것이다. 각 하위 문제의 해결을 계산한 뒤, 그 해결책을 저장하여 후에 같은 하위 문제가 나왔을 경우 그것을 간단하게 해결할 수 있다. 이러한 방법으로 동적 계획법은 계산 횟수를 줄일 수 있다. 특히 이 방법은 하위 문제의 수가 기하급수적으로 증가할 때 유용하다.

동적 계획 알고리즘은 최단 경로 문제, 행렬의 제곱 문제 등의 최적화에 사용된다. 이것은 동적 계획법은 문제를 해결하기 위한 모든 방법을 검토하고, 그 중에 최적의 풀이법을 찾아내기 때문이다. 이에 우리는 동적 계획법을 모든 방법을 일일이 검토하여 그 중 최적 해를 찾아내는 주먹구구식 방법이라고 생각할 수 있다. 그러나 문제가 가능한 모든 방법을 충분히 빠른 속도로 처리할 수 있는 경우, 동적 계획법은 최적의 해법이라고 말할 수 있다.

때로는 단순한 재귀 함수에 저장 수열(이전의 데이터를 모두 입력하는 수열)을 대입하는 것 만으로도 최적 해를 구할 수 있는 동적 알고리즘을 찾을 수 있다. 그러나 대다수의 문제는 이보다 훨씬 더 복잡한 프로그래밍을 요구한다. 그 중에 일부는 여러 개의 매개 변수를 이용하여 재귀 함수를 작성해야 하는 것도 있고, 아예 이러한 방법으로 동적 알고리즘을 짤 수 없는 문제 또한 존재한다. 이번 실험에서는 이러한 동적 계획법을 사용하여 연속 행렬 곱셈과 이진 탐색 트리의 최적의 트리 구조를 구하는 방법에 대해서 실험하고자 한다.

1. 연속 행렬 곱셈(Chained Matrix Multiplication)의 단계적 해석

주어진 연속된 행렬의 순서대로 행과 열을 중복되는 값을 제외하고 저장한 배열을 D, 주어진 행렬들을 i번째 위치의 행렬부터 j번째 위치의 행렬까지의 최소 곱셈 횟수를 저장한 배열을 M, 주어진 행렬들을 i번째 행렬에서 j번째 행렬까지 곱셈을 할 때 최소의 곱셈 횟수가 되기 위해 분할해야 하는 행렬의 위치를 저장한 배열이 P이다. 이번 실험에서는 행렬 5개([10 * 4], [4 * 5], [5 * 20], [20 * 2], [2 * 5])일때 최적의 행렬 곱셈을 찾아내는 방법을 알아보겠다.



[그림 1] 연산 전 행렬 W, D, P

```
void chainMatrixMul(void) {
    int i, j, diagonal, k, mul_count;
    // diagonal == 행렬 대각선
    for(diagonal = 1; diagonal < MAX; diagonal++) {
        // i = 시작 행렬
        for(i = 0; i < MAX - diagonal; i++) {
          // j = 끝행렬
           j = i + diagonal;
           M[i][j] = INF;  // 대각선 아래를 INF로 초기화
           // k = 중간 행렬
           for(k = i; k < j; k++) {
              // 행렬의 곱셈 횟수 A0.행 * A0.열 == A1.행 * A1.열
              mul_count = M[i][k] + M[k + 1][j] + D[i] * D[k + 1] * D[j + 1];
              if(mul_count < M[i][j]) {
                 // 최소 곱셈 연산 횟수 할당
                 M[i][j] = mul_count;
                 // i ~ j까지의 최소곱을 하기 위한 중간 행렬 지점 지정
                 P[i][j] = k;
              }
           }
        }
        printf("\n");
    }
}
```

[코드 1] chainMatrixMul( )

[코드 1]을 살펴보면, 3중 for문을 사용한다. 여기서 diagonal은 시퀀스 마다 행렬의 대각선을 기준으로 대각선 모양으로 결과값이 누적된다. 그리고 i, j, k는 행렬의 인덱스를 의미한다. 주어진 주어진 행렬의 개수는 총 5개이므로, diagonal이 동작하는 횟수는 총 4번이다. 그리고 시작행렬 i는 diagonal 즉 시퀀스가 증가할 때마다, 최적 해가 누적되므로 점차 범위가 감소하고, 연산 범위가 감소하고, k는 연산 시 분리해야하는 행렬을 i 반복문의 시퀀스마다 탐색하고, j는 행렬 곱셈의 연산 범위를 지정한다.

이러한 시퀀스마다 행렬 곱셈의 최적해를 구하는 코드는 조건문으로부터 시작된다. mul_count는 행렬 곱셈의 횟수를 저장한 변수이다. 이러한 곱셈의 횟수가 최소 곱셈의 횟수가 저장된 M에서 비교했을 때, 최초 코드 동작 시에는 M[i][j]를 무한(INF)로 지정한다. 다음으로 중간 행렬을 탐색할 때마다 최솟값이 저장된 M과 현재 탐색중인 행렬 곱셈의 횟수와 비교해서 최소 값을 M[i][j]에, 최소 값인 행렬 곱셈의 중간 행렬의 위치 k를 P[i][j]에 저장한다.

즉 작은 문제인 두개의 행렬 곱부터 시작해서 점차 곱하는 행렬의 개수를 늘려가는 동적 계획법을 적용한 알고리즘이라고 할 수 있다. 이 코드를 단계적으로 연산 과정을 출력해 보면 [그림 2]와 같다.

```
diagonal = 1
    i = 0, j = 1, k = 0, A * B = 200
    M[0][1] = 200, P[0][1] = 0
    i = 1, j = 2, k = 1, B * C = 400
    M[1][2] = 400, P[1][2] = 1
    i = 2, j = 3, k = 2, C * D = 200
    M[2][3] = 200, P[2][3] = 2
    i = 3, j = 4, k = 3, D * E = 2000
    M[3][4] = 2000, P[3][4] = 3

diagonal = 2
    i = 0, j = 2, k = 0, A * C = 1200
    M[0][2] = 1200, P[0][2] = 0
    i = 0, j = 2, k = 1, A * C = 1200
    i = 1, j = 3, k = 1, B * D = 240
    M[1][3] = 240, P[1][3] = 1
    i = 1, j = 3, k = 2, B * D = 560
    i = 2, j = 4, k = 2, C * E = 7000
    M[2][4] = 7000, P[2][4] = 2
    i = 2, j = 4, k = 3, C * E = 700
    M[2][4] = 700, P[2][4] = 3

diagonal = 3
    i = 0, j = 3, k = 0, A * D = 320
    M[0][3] = 320, P[0][3] = 0
    i = 0, j = 3, k = 1, A * D = 500
    i = 0, j = 3, k = 2, A * D = 1600
    i = 1, j = 4, k = 1, B * E = 1700
    M[1][4] = 1700, P[1][4] = 1
    i = 1, j = 4, k = 2, B * E = 6400
    i = 1, j = 4, k = 3, B * E = 640
    M[1][4] = 640, P[1][4] = 3

diagonal = 4
    i = 0, j = 4, k = 0, A * E = 2640
    M[0][4] = 2640, P[0][4] = 0
    i = 0, j = 4, k = 1, A * E = 3400
    i = 0, j = 4, k = 2, A * E = 13200
    i = 0, j = 4, k = 3, A * E = 1320
    M[0][4] = 1320, P[0][4] = 3
```

[그림 2] chainMatrixMul( )의 단계적 결과

이러한 연산이 모두 완료 된 뒤, 행렬 M과 P를 보면 [그림 3]과 같다. 그리고 이러한 결과를 통해 도출할 수 있는 최적의 곱셈 순서는 (A(B(DC)))E이다.

```
Matrix M:
    0     200    1200     320    1320
    0       0     400     240     640
    0       0       0     200     700
    0       0       0       0    2000
    0       0       0       0       0

Matrix P:
    0       0       0       0       3
    0       0       1       1       3
    0       0       0       2       3
    0       0       0       0       3
    0       0       0       0       0
```

[그림 3] 연산 후 행렬 M, P

2. 이진 탐색 트리의 최적 트리 구조의 단계적 해석

코드를 보기에 앞서 먼저 이진 탐색 트리의 특성을 살펴보자.

- 하나의 노드는 2개의 자식을 가질 수 있다.
- 각 노드는 하나의 Key값만 가질 수 있다.
- 왼쪽 트리는 부모 노드보다 Key값이 작다.
- 오른쪽 트리는 부모 노드보다 Key값이 크다.

트리는 깊이(Depth)를 가지고 있는데, 이러한 깊이가 깊어질 수록 비교해야 하는 횟수가 증가하게 된다. 이 때 우리는 Key값을 찾기 위한 확률이 존재하는데, 트리의 확률이 작으면 작을수록 좋다. 그런데 깊이가 깊어질수록 찾기위한 비교 횟수가 증가하므로, 확률 * 깊이가 된다. 이때 이러한 확률을 저장한 배열을 P, 주어진 Key를 저장한 배열을 KEY이다. 그리고 최적의 트리가 되기 위해 루트가 되어야하는 Key를 저장한 행렬을 R, 루트 노드가 변화하고 노드의 개수가 증가할 때마다의 트리의 확률을 저장한 행렬을 A라고 하자. KEY, P, R, A의 초기화된 값은 [그림 4]와 같다. 이때 R과 A의 경우 행의 인덱스가 1부터 시작한다.

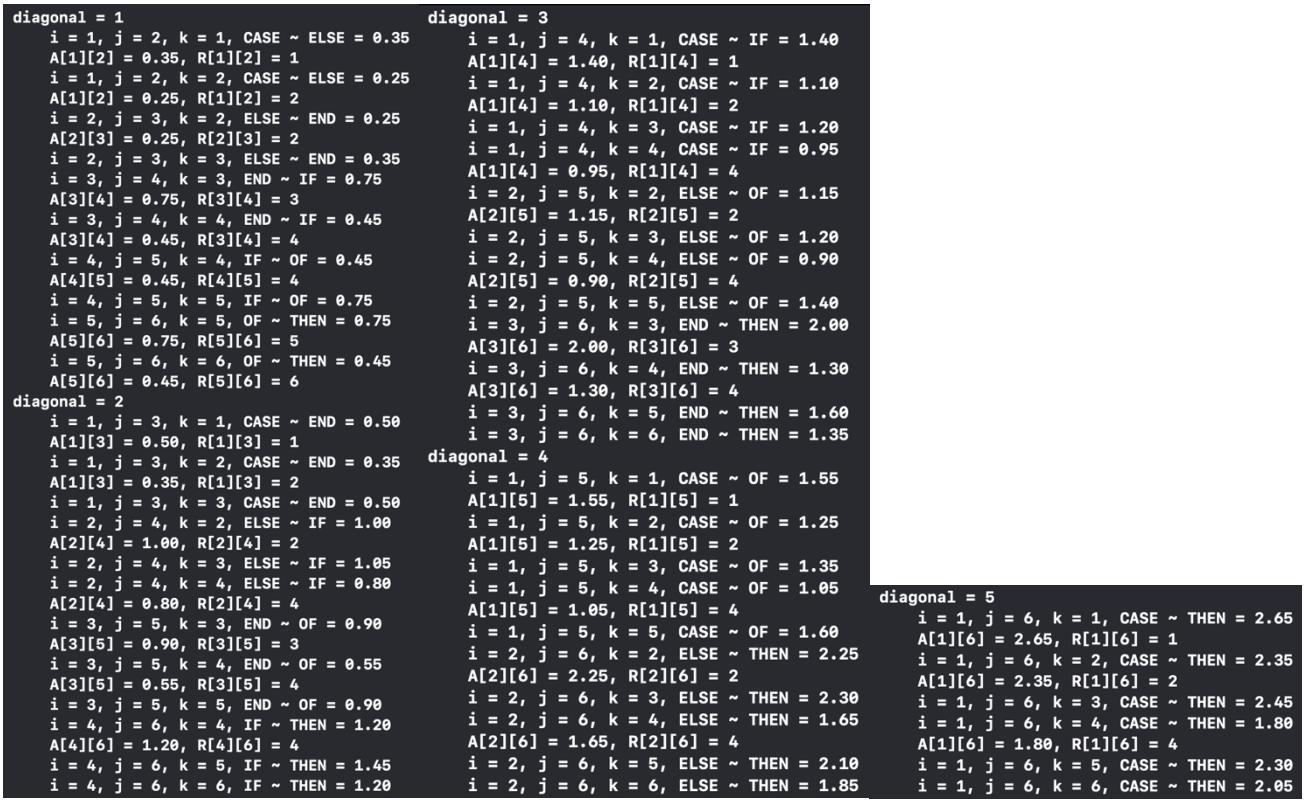
```
Matrix KEY:
CASE    ELSE    END IF  OF  THEN
Matrix P:
0.05    0.15    0.05    0.35    0.05    0.35    0.00
Matrix R:
0   1   0   0   0   0   0
0   0   2   0   0   0   0
0   0   0   3   0   0   0
0   0   0   0   4   0   0
0   0   0   0   0   5   0
0   0   0   0   0   0   6
0   0   0   0   0   0   0

Matrix A:
0.00    0.05    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.15    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.05    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.35    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.05    0.00
0.00    0.00    0.00    0.00    0.00    0.00    0.35
0.00    0.00    0.00    0.00    0.00    0.00    0.00
```

[그림 4] 연산 전 행렬 KEY, P, R, A

```
void optimalSearch(void) {
    int i, j, k, diagonal;
    float comp_p;

    for(int i = 1; i <= MAX; i++) {
        A[i][i] = P[i];
        A[i][i-1] = 0;
        R[i][i] = i;
        R[i][i-1] = 0;
    }
    A[MAX][MAX-1] = 0;
    R[MAX][MAX-1] = 0;
    // diagonal = 행렬 대각선
    for(diagonal = 1; diagonal <= MAX; diagonal++) {       // i = 시작 위치
        for(i = 1; i <= MAX - diagonal; i++) {
            // j = 끝 위치
            j = i + diagonal;
            A[i][j] = INF;
            // k = 중간 위치
            for(k = i; k <= j; k++) {
                comp_p = A[i][k - 1] + A[k + 1][j] + sum(i, j);
                if(comp_p < A[i][j]) {
                    A[i][j] = comp_p;    // 최소 곱셈 연산 횟수 할당
                    R[i][j] = k;          // i ~ j까지의 최소곱을 하기 위한 중간 행렬 지점 지정
                }
            }
        }
    }
}

float sum(int start, int end) {
    float sum = 0;
    for(int i = start; i <= end; i++) {
        sum += P[i];
    }
    return sum;
}
```

[코드 2] optimalBST( ), sum( )

[코드 2]를 보면 먼저 A행렬과 R행렬을 초기화 시켜준다. 이 때 초기화 시에 대각선을 기준으로 각 순서를 차례대로 집어넣는 것을 볼 수 있다. 그리고 대각선을 기준으로 아래는 0으로 초기화하는데 이는 대각선 기준 아래를 사용하지 않는다는 것으로 해석할 수 있다. 이전에 본 연속 행렬 곱셈과 마찬가지로 최적 이진 탐색 트리를 구하기 위한 과정은 3중 for문으로 구성되어있다. diagonal은 행렬의 대각선을, i는 시작 위치, j는 끝 위치, k는 중간 위치를 의미한다. 여기서도 각 시퀀스마다 결과값을 조건문에서 저장하게 되는데 comp_p는 k위치의 Key를 기준으로 왼쪽과 오른쪽의 트리의 확률의 합과 Key i번째부터 j번째까지의 Key의 확률들의 합을 담은 변수이다. 즉 k위치의 Key가 루트 노드일 때 왼쪽과 오른쪽 트리가 최적의 트리 구조일 때를 말한다. 행렬 A[i][j]에 저장된 값을 처음에는 INF로 초기화 시켜주고 k의 위치 즉 루트 노드의 Key값을 차례대로 변화 시켜보면서 트리의 확률이 최소인 값을 M[i][j]에 저장하고 위치 k를 R[i][j]에 저장하게 된다. 이 또한 마찬가지로 두개의 Key를 가진 최적의 트리를 시작으로 점진적으로 Key의 개수를 증가시키며 결국 n개의 Key를 가진 최적의 트리를 도출해내는 동적 계획법을 적용한 알고리즘임을 알 수 있다. 이 코드를 단계적으로 출력해보면 [그림 5]와 같다.



[그림 5] optimalBST( )의 단계적 결과

연산을 모두 마치고 난 후 행렬 R과 A는 [그림 6]과 같다.

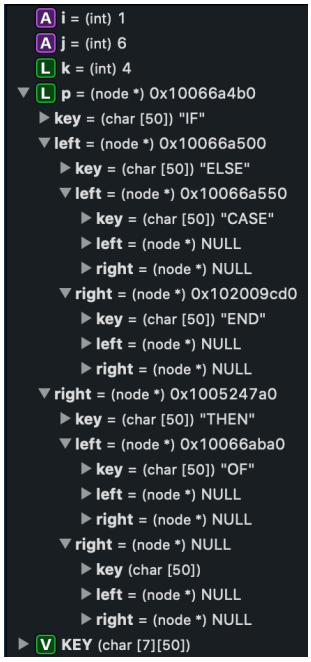
```
Matrix R:
0   1   2   2   4   4   4
0   0   2   2   4   4   4
0   0   0   3   4   4   4
0   0   0   0   4   4   4
0   0   0   0   0   5   6
0   0   0   0   0   0   6
0   0   0   0   0   0   0

Matrix A:
0.00    0.05    0.25    0.35    0.95    1.05    1.80
0.00    0.00    0.15    0.25    0.80    0.90    1.65
0.00    0.00    0.00    0.05    0.45    0.55    1.30
0.00    0.00    0.00    0.00    0.35    0.45    1.20
0.00    0.00    0.00    0.00    0.00    0.05    0.45
0.00    0.00    0.00    0.00    0.00    0.00    0.35
0.00    0.00    0.00    0.00    0.00    0.00    0.00
```

[그림 6] 연산 후 행렬 R, A

이 행렬 R을 통해 4번째 Key값인 IF가 루트 노드일 때 최적의 이진 탐색 트리를 가지고 있음을 알 수 있다. 그렇다면 이 결과를 통해 [코드 3]을 가지고 이진 탐색 트리를 구성해보면 다음과 같다.

```c
typedef struct node {
    char key[STRING_MAX];
    struct node* left;
    struct node* right;
}node;

node* tree(int i, int j) {  // 이진트리 생성 함수
   int k;
   k = R[i][j];
   node* p = malloc(sizeof(node));

   if (k == 0) return NULL;
   else {
      strcpy(p->key, KEY[k]);
      p->left = tree(i, k - 1);
      p->right = tree(k + 1, j);
      return p;
   }
}

void search(node* root, char* keyin) {
    int found;
    node* cursor;

    cursor = root;  // cursor = 현재 위치 노드
    found = FALSE;  // found = 원하는 key값을 찾았는지 여부
    while(found != TRUE) {
       // 원하는 key값을 못찾았을 경우
       if(cursor->key[0] == '\0') return;
       // 원하는 key값을 찾았을 경우
       else if(strcmp(cursor->key, keyin) == 0) found = TRUE;
       // 주어진 key값이 더 작을 경우
       else if (strcmp(cursor->key, keyin) == 1) cursor = cursor->left;
       // 주어진 key값이 더 클 경우
       else cursor = cursor->right;
    }
}
```

[코드 3] 변수 node, tree( ), search( )



[그림 7] 루트 노드가 IF인 최적 이진 탐색 트리 구조

3. ChainedMatrixMultiplication.c

```
#include <stdio.h>

#define min(x,y) (((x) < (y)) ? (x) : (y))
#define INF 99999
#define MAX 5

void printOrder(int i,int j);
void printMatrix(void);

// D[곱하려는 행렬 개수 + 1] = {A0.row, A0.column == A1.row, A1.column == A2.row ... }
// A0 = 10 * 4, A1 = 4 * 5
int D[MAX + 1] = {10, 4, 5, 20, 2, 50};
// M[곱하려는 행렬 개수][곱하려는 행렬 개수]
// M[i][j] = 주어진 행렬 중 i ~ j 까지의 곱셈시 필요한 최소 곱셈 연산 횟수
int M[MAX][MAX] = {0};
// P[i][j] = i ~ j까지의 곱셈시 최소곱이 되기위해 거쳐야 하는 중간 행렬 지점
int P[MAX][MAX] = {0};

int main(void) {
   int i, j, diagonal, k, mul_count;
   for(diagonal = 1; diagonal < MAX; diagonal++) { // diagonal == 행렬 대각선
      for(i = 0; i < MAX - diagonal; i++) {      // i = 시작 행렬
         j = i + diagonal;  // j = 끝행렬, 0~1, 1~2, 2~3, 3~4 / 0~2, 1~3, 2~4 / 0~3, 1~4 / 0~4
         M[i][j] = INF;  // 대각선 아래를 INF로 초기화
         for(k = i; k < j; k++) {   // k = 중간 행렬
            // 행렬의 곱셈 횟수 A0.row * A0.column == A1.row * A1.column
            mul_count = M[i][k] + M[k + 1][j] + D[i] * D[k + 1] * D[j + 1];
            if(mul_count < M[i][j]) {
               M[i][j] = mul_count;   // 최소 곱셈 연산 횟수 할당
               P[i][j] = k;          // i ~ j까지의 최소곱을 하기 위한 중간 행렬 지점 지정
            }
         }
      }
      printf("\n");
   }
   printOrder(0,MAX-1);
   return 0;
}

void printMatrix(void) {
   printf("\n\tMatrix M:\n");
   for(int i = 0; i < MAX; i++) {
      for(int j = 0; j < MAX; j++) {
         printf("\t%5d", M[i][j]);
      }
      printf("\n");
   }
   printf("\n\tMatrix P:\n");
   for(int i = 0; i < MAX; i++) {
      for(int j = 0; j < MAX; j++) {
         printf("\t%5d", P[i][j]);
      }
      printf("\n");
   }
   printf("\n");
}

void printOrder(int i,int j) {
   if(i == j) printf("%c", 'A' + i);
   else {
      int k = P[i][j];
      printf("(");
      printOrder(i,k);
      printOrder(k + 1,j);
      printf(")");

   }
```

```
}
```

4. OptimalBST.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

#define MAX 6
#define STRING_MAX 50
#define INF 999

// KEY = 주어진 Key값
char KEY[MAX + 1][STRING_MAX] = {"\0", "CASE", "ELSE", "END", "IF", "OF", "THEN"};
// P = 각 Key마다의 확률 값
float P[MAX + 1] = {INF, 0.05, 0.15, 0.05, 0.35, 0.05, 0.35};
// R = 주어진
int R[MAX + 1][MAX + 1] = {0};
float A[MAX + 1][MAX + 1] = {0};

typedef struct node {
   char key[STRING_MAX];
   struct node* left;
   struct node* right;
}node;

void printMatrix(void);
void initMatrix(void);
void search(node* root, char* keyin);
void optimalBST(void);
float sum(int start, int end);

node* tree(int i, int j)  // 이진트리 생성 함수
{
  int k;
  k = R[i][j];
  node* p = malloc(sizeof(node));

  if (k == 0)
    return NULL;
  else {
    strcpy(p->key, KEY[k]);
    p->left = tree(i, k - 1);
    p->right = tree(k + 1, j);
    return p;
  }
}

int main(void) {
   optimalBST();
   tree(1, MAX);
   return 0;
}

void search(node* root, char* keyin) {
   int found;
   node* cursor;

   cursor = root;  // cursor = 현재 위치 노드
   found = FALSE;  // found = 원하는 key값을 찾았는지 여부
   while(found != TRUE) {
      // 원하는 key값을 못찾았을 경우
      if(cursor->key[0] == '\0')
         return;
      // 원하는 key값을 찾았을 경우
      else if(strcmp(cursor->key, keyin) == 0)
         found = TRUE;
```

```
        // 주어진 key값이 더 작을 경우
        else if (strcmp(cursor->key, keyin) == 1)
            cursor = cursor->left;
        // 주어진 key값이 더 클 경우
        else cursor = cursor->right;
    }
}

void optimalBST(void) {
    int i, j, k, diagonal;
    float comp_p;

    for(int i = 1; i <= MAX; i++) {
        A[i][i] = P[i];
        A[i][i-1] = 0;
        R[i][i] = i;
        R[i][i-1] = 0;
    }
    A[MAX][MAX-1] = 0;
    R[MAX][MAX-1] = 0;
    // diagonal = 행렬 대각선
    for(diagonal = 1; diagonal <= MAX; diagonal++) {
        // i = 시작 위치
        for(i = 1; i <= MAX - diagonal; i++) {
            // j = 끝 위치
            j = i + diagonal;
            A[i][j] = INF;
            // k = 중간 위치
            for(k = i; k <= j; k++) {
                comp_p = A[i][k - 1] + A[k + 1][j] + sum(i, j);
                if(comp_p < A[i][j]) {
                    A[i][j] = comp_p;    // 최소 곱셈 연산 횟수 할당
                    R[i][j] = k;          // i ~ j까지의 최소곱을 하기 위한 중간 행렬 지점 지정
                }
            }
        }
    }
}

float sum(int start, int end) {
    float sum = 0;
    for(int i = start; i <= end; i++)
        sum += P[i];
    return sum;
}

void initMatrix(void) {
    for(int i = 0; i < MAX; i++) {
        A[i][i] = P[i];
        R[i][i] = i;
    }
}

void printMatrix(void) {
    printf("\n\tMatrix KEY:\n");
    for(int i = 1; i <= MAX + 1; i++) {
        printf("\t%s", KEY[i]);
    }
    printf("\n\tMatrix P:\n");
    for(int i = 1; i <= MAX + 1; i++) {
        printf("\t%.2f", P[i]);
    }
    printf("\n\tMatrix R:\n");
    for(int i = 1; i <= MAX + 1; i++) {
        for(int j = 0; j < MAX + 1; j++) {
            printf("\t%d", R[i][j]);
        }
        printf("\n");
    }
    printf("\n\tMatrix A:\n");
```

```
    for(int i = 1; i <= MAX + 1; i++) {
        for(int j = 0; j < MAX + 1; j++) {
            printf("\t%.2f", A[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
```