10 Algorithm Design. Dynamic Programming and Heuristics

10.1 Objectives

The learning objectives of this laboratory session are:

- To understand the challenges of implementing *dynamic programming* algorithm development method.
- To get hands-on experience with implementing dynamic programming algorithms.
- To develop critical thinking concerning implementation decisions algorithms which use a dynamic programming approach.

The outcomes for this session are:

- Improved skills in C programming.
- A better understanding of dynamic programming algorithms.
- The ability to make implementation decisions concerning dynamic programming algorithms.
- The ability to decide when dynamic programming is suitable for solving a problem.

.

10.2 Brief Theory Reminder

10.2.1 Dynamic Programming

Dynamic programming (DP) is a method of algorithm development which is suited for finding optimal solutions. The solution is found by making a sequence of decisions which depend on previously made decisions, and all satisfy the principle of optimality. This principle may be stated as follows:

Let $\langle s_0, s_1, \ldots, s_n \rangle$ be a sequence of states, where s_0 is the initial state and s_n is the final state. The final state is reached by making a sequence of decisions, $d_1, d_2, \ldots, d_n$, as suggested by Figure 10.1.

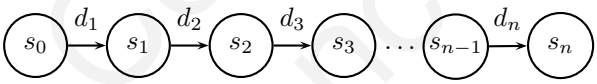


Figure 10.1: States and decisions.

If the sequence of decisions $\langle d_i, d_{i+1}, \ldots, d_j \rangle$ which changes state s_{i-1} into state s_j (with intermediate states $s_i, s_{i+1}, \ldots, s_{j-1}$) is optimal, and if for all $i \leq k \leq j-1$ both $\langle d_i, d_{i+1}, \ldots, d_k \rangle$ and $\langle d_{k+1}, d_{k+2}, \ldots, d_j \rangle$ are optimal sequences of decisions which transform state s_{i-1} into s_k, and state s_k into s_j respectively, then the principle of optimality is satisfied.

To apply this method we may proceed as follows:

- Check that the principle of optimality is satisfied.
- Write the recurrence equations which result from the rules which govern the state changes and solve them.

In CLRS[1], the authors suggest that the development of a dynamic programming algorithm can be broken into a sequence of following four steps:

1. Characterize the structure of an optimal solution. That is, decompose the problem into smaller problems, and find a relation between the structure of the optimal solution of the original problem and the solutions of the smaller problems

[1] "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, 3rd edition, MIT Press, 2009

2. Recursively define the value of an optimal solution – express the solution of the original problem in terms of optimal solutions for smaller problems
3. Compute the value of an optimal solution in a bottom-up fashion, by using a table structure.
4. Construct an optimal solution from computed information.

Note that the last two steps may often be combined.
The CLRS also presented an alternative approach called **memoization**. It works as follows:

- **Store**, don't recompute.
- Make a table indexed by subproblem.
- When solving a subproblem:
  - **Lookup** in the table.
  - If answer is there, **use it**.
  - Otherwise, **compute** answer, then **store** it.

Example 1. Matrix Chain Multiplication

Consider the multiplication of a sequence of matrices

$$R = A_1 \times A_2 \times \ldots \times A_n,$$

where A_i with $1 \leq i \leq n$ is of size $d_i \times d_{i+1}$. The resulting matrix, R, will be of size $d_1 \times d_{n+1}$. It is common knowledge that when multiplying two matrices, say A_i and A_{i+1} we must effect $d_i \times d_{i+1} \times d_{i+2}$ multiplications. Remember that matrix multiplication is not commutative. If the matrices to be multiplied have different numbers of rows/columns, then the number of operations used to get R depends on the order of performing the multiplications. What we would like to find is the order of performing these multiplications which results in a minimum number of operations.
Let C_{ij} be the minimum number of multiplications for calculating the product $A_i \times A_{i+1} \times \ldots \times A_j$ for $1 \leq j \leq n$. Note that:

- $C_{i,i} = 0$, i.e., multiplication for a chain with only one matrix is done at no cost.
- $C_{i,i+1} = d_i \times d_{i+1} \times d_{i+2}$.
- $C_{1,n}$ will be the minimum value.
- The principle of optimality is observed, i.e.

  $$C_{i,j} = \min\{C_{i,k} + C_{k+1,j} + d_i \times d_{k+1} \times d_{j+1}\},$$

  for $i \leq k < j$, and the associations are $(A_i \times A_{i+1} \times \ldots \times A_k) \times (A_{k+1} \times A_{k+2} \times \ldots \times A_j)$.

We have to compute $C_{i,i+d}$ for each level d till we get $C_{1,n}$. In order to build the binary tree which describes the order of multiplications, we will also retain the value of k which was used in obtaining the minimum, which enables us to show how the matrices are associated. The vertices of this tree will contain the limits of the matrix subsequence for which the matrices are associated. The root will be $(1, n)$, and a subtree rooted at (i, j) will have (i, k) and $(k + 1, j)$ as children, where k is the value for which the optimum is obtained. Listing 10.1 has C code implementing optimal matrix multiplication.

Listing 10.1: An implementatrion of matrix chain multiplication using DP

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

#define  MAXN 10

typedef struct _NodeT
{
    long ind1, ind2;
    struct _NodeT *left,*right;
}
NodeT;

void matMulOrder( long C[ MAXN ][ MAXN ], int D[ MAXN + 1 ], int n )
/* Determines optimal multiplication order for a sequence of matrices
   A1 × A2 × ... × An of sizes D1 × D2, D2 × D3, ..., Dn × Dn+1 */
{
    int i, j, k, l, pos;
    long min, q;

    for( i = 1; i <= n; i++ ) C[ i ][ i ] = 0;
    for(l = 1; l <= n - 1; l++ )
        for( i = 1; i <= n - l; i++ )
```

```
        {
            j = i + l;
            min = LONG_MAX;
            for( k = i; k <= j - 1; k++ )
            {
                q = C[ i ][ k ] + C[ k + 1 ][ j ] + (long) D[ i ] * D[ k + 1 ] * D[ j + 1];
                if( q < min)
                {
                    min = q;
                    pos = k;
                }
            }
            C[ i ][ j ] = min;
            C[ j ][ i ] = pos;
        }
}
NodeT *buildTree( NodeT *p, int i, int j, long C[ MAXN ][ MAXN ] )
{
    p = ( NodeT * )malloc( sizeof( NodeT ));
    p->ind1 = i;
    p->ind2 = j;
    if( i < j )
    {
        p->left = buildTree( p->left, i, C[ j ][ i ], C ) ;
        p->right = buildTree( p->right, C[ j ][ i ] + 1, j, C );
    }
    else
    {
        p->left = p->right= NULL;
    }
    return p;
}
void postOrder( NodeT *p, int level )
{
    int i;
    if( p != NULL )
    {
        postOrder( p->left, level + 1 );
        postOrder( p->right, level + 1 );
        for( i = 0; i <= level; i++) printf("  ");
        printf( "(%ld, %ld)\n", p->ind1, p->ind2 );
    }
}
void main(void)
{
    int i, j, n;
    long C[ MAXN ][ MAXN ];
    int D[MAXN+1]; /* matrix dimensions */
    NodeT *root = NULL;

    printf("\nInput no. of matrices: ");
    scanf( "%d", &n );
    while ( getchar() != '\n' );
    printf("\nMatrix dimensions\n");
    for( i = 1; i <= n + 1; i++ )
    {
        printf("\nEnter dimension D[%d]=", i );
        scanf( "%d", &D[ i ] );
        while ( getchar() != '\n' );
    }
    matMulOrder( C, D, n );
    /* computed cost matrix C */
    printf("\nCost matrix C\n");
    for(i = 1; i <= n; i++ )
    {
        for( j = 1; j <= n; j++)
            printf( "%6ld", C[ i ][ j ] );
        printf( "\n" );
    }
    printf("\nMinimum no. of multiplications = %ld", C[ 1 ][ n ] );
    while ( getchar() != '\n' );
    root = buildTree( root, 1, n, C );
    printf("\nPostorder traversal of tree\n");
    postOrder( root, 0 );
```

```
    while ( getchar() != '\n' );
}
```

Example 2. 1-0 Knapsack

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. If the weights are integer values, then we have two integer arrays $v[0..n-1]$ and $w[0..n-1]$ which represent values and weights associated with the n items respectively. An integer W represents knapsack capacity. We have to find out the maximum value subset of v_i, $1 \leq i \leq n$ such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

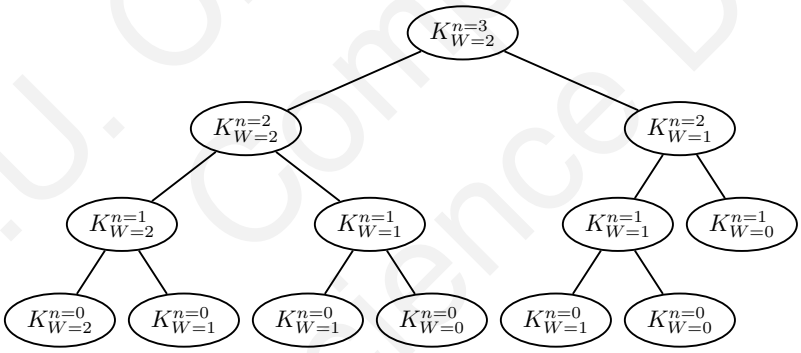
A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

1. **Optimal Substructure**:
To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set. Therefore, the maximum value that can be obtained from n items is max of following two values:

1. Maximum value obtained by $k-1$ items and W weight (excluding k^{th} item).
2. Value of k^{th} item + maximum value obtained by $k-1$ items and $W-w_k$ (the weight of the k^{th} item $-$ including k^{th} item).

If the weight of k^{th} item is greater than W, then the k^{th} item cannot be included and case 1 is the only possibility.

2. **Overlapping Subproblems**
The 0-1 Knapsack problem also has the overlapping subproblem property, since subproblems are evaluated again, of a dynamic programming problem. You can easily see this if you draw a recursion tree, as shown in Figure 10.2. Recomputations of the same subproblems can be avoided by storing solutions in a temporary array K, in a bottom up manner.



The recursion tree above is for a knapsack of capacity $W = 2$, and the items described in the table below. The two parameters shown in every node of the recursion tree above are n (number of items) and W (weight of knapsack).

item #	1	2	3
weight ($w[]$)	1	1	1
value ($v[]$)	10	20	30

Figure 10.2: Example recursion tree from a Knapsack problem instance.

Listing 10.2 shows an implementation of a dynamic programming solution for the 0-1 knapsack problem. Note that it works for small values of W, w, v, and n. For large values allocation of arrays on the stack is not feasible.

Listing 10.2: Solution to the 0-1 Knapsack problem.

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

typedef struct
```

```
{
    int value; // item value
    int weight; // item weight
} ItemT;

enum {DONT_TAKE=0, TAKE=1};

// Returns maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}
// allocate a rows by cols matrix of ints
int **allocMatrix(int rows, int cols)
{
    int **matrix = (int **)malloc( rows * sizeof(int *) );
    if (matrix)
    {
        matrix[0] = (int*) malloc( rows  * cols * sizeof(int) );
        for(int i = 1; i < rows; ++i)
        {
            matrix[i] = matrix[i - 1] + cols;
        }
    }
    return matrix;
}
void printMatrix(const char *msg, int **m, const int r, const int c)
{
    printf("%s\n", msg);
    for (int i=0; i <= r; i++)
    {
        for (int j=0; j <= c; j++)
        {
            printf("%02d ", m[i][j]);
        }
        printf("\n");
    }
}
// Returns the maximum value that can be put in a knapsack of capacity W
int solveKnapsack(int W, int n, ItemT item[], int **taken)
{
    int **K = allocMatrix(n+1, W+1);
    // Build table K[][] in bottom up manner
    for (int w = 0; w <= W; w++)
    {
        K[0][w] = 0; // no item
        taken[0][w] = 0;
    }
    for (int i = 0; i <= n; i++)
    {
        K[i][0] = 0; // no weight (capacity)
        taken[i][0] = 0;
    }
    for (int i = 1; i <= n; i++)
    {   // all items
        for (int w = 0; w <= W; w++)
        {   // all possible knapsack capacities
            if (item[i].weight <= w)
            {   // item i may be taken
                K[i][w] = max(K[i-1][w], item[i].value + K[i-1][w - item[i].weight]);
                if (K[i-1][w] >= K[i][w])
                    taken[i][w] = DONT_TAKE;
                else
                    taken[i][w] = TAKE;
            }
            else
            {   // leave item i
                K[i][w] = K[i-1][w];
                taken[i][w] = DONT_TAKE; // 0
            }
        }
    }

    int optimum = K[n][W];
```

```
    printf("n=%d W=%d\n\n", n, W);
    printMatrix("K:", K, n, W);
    printMatrix("taken:", taken, n, W);
    free(K[0]);
    free(K);
    return optimum;
}
// recursive function for printing taken items
void recPrintKnapsackItems(int n, int w, int **taken, ItemT items[])
{
    if (n >= 0)
    {
        if (taken[n][w] == TAKE)
        {
            printf("%d [%d %d] ", n, items[n].value, items[n].weight);
            w -= items[n].weight;
        }
        recPrintKnapsackItems(n-1, w, taken, items);
    }
    printf("\n");

}
// non-recursive function for printing taken items
void printKnapsackItems(int n, int w, int **taken, ItemT items[])
{
    printf("Items taken: ");
    while (n >= 0)
    {
        if (taken[n][w] == TAKE)
        {
            printf("%d [%d %d] ", n, items[n].value, items[n].weight);
            w -= items[n].weight;
        }
        n--;
    }
    printf("\n");
}
int main()
{
    ItemT items[] =
    {
        // value, weight pairs
        // {60, 10}, {100, 20}, {120,30}
        {0, 0}, {10, 5}, {40, 4}, {30, 6}, {50, 3}
    };
    int  W = 10; // knapsack capacity
    int n = sizeof(items)/sizeof(items[0]) - 1;
    int **taken = allocMatrix(n+1, W+1);
    printf("%d\n", solveKnapsack(W, n, items, taken));
    printKnapsackItems(n, W, taken, items);
    // recPrintKnapsackItems(n, W, taken, items);
    free(taken[0]);
    free(taken);
    return 0;
}
```

10.2.2 Heuristics

An *heuristic* algorithm is an algorithm which gives an approximate solution, not necessarily optimal, but which can be easily implemented and has a polynomial growth rate. Heuristics are used in solving problems where a global optimum is not known to exist and where results close to optimum are acceptable. One way to cope with such problems is to decompose the process of finding a solution into successive processes for which the optimum is searched. Generally, this does not lead to a global optimum, as local optima do not imply the global optimum is reached. In practice, a number of approximate solutions are found, and the best of them is then selected.

E.g. Let us now consider the traveling salesman problem. Let $G = (X, \Gamma)$ be an undirected complete[2] graph with a strictly positive cost associated to each edge. We are required to find a cycle, starting with vertex i, which passes through all nodes exactly once, and ends at i. It is possible to find an optimal solution, using backtracking, but that will take exponential time.

[2] remember: complete=all edges are present

The heuristic for which the code is given below, uses the greedy method and requires polynomial time.
Let $(v_1, v_2, \ldots, v_k)$ be an already built path. The steps to build the solution are:

1. If $\{v_1, v_2, \ldots, v_k\} = X$ then add edge (v_k, v_1) and the cycle is completed.
2. If $\{v_1, v_2, \ldots, v_k\} \neq X$ then add the minimum cost edge which connects v_k to a vertex in X not included yet.

As a cycle is a closed path, we may start at any one node. So we could pick up each of $1, 2, \ldots, n$, in turn, as a start vertex, find the cycle, and retain the minimum of them all.

Example. Traveling Salesman Problem.

A C implementation for this problem is shown in Listing 10.3.

Listing 10.3: Solution to the traveling salesman problem.

```
#include <stdio.h>
#include <limits.h>

#define MAXN 10
#define INFTY INT_MAX

/* Finds traveling salesman tour
   n = number of nodes
   C = matrix of costs
   i = start vertex
   tour = vector containing the vertices of the tour
   cost = cost of the tour */
void TravSMan( int n, int c[ MAXN ][ MAXN ],
               int i, int tour[ MAXN+1 ],
               int *cost )
{
    int p[ MAXN ];
    int v, vmin;
    int costmin;

    for(int k = 1; k <= n; k++ )
        p[k] = 0;
    *cost = 0;
    p[ i ] = 1;
    tour[ 1 ] = i;
    v = i; /* current node */
    for (int k = 1; k < n; k++ )
    {
        /* add n-1 edges, one by one */
        costmin = INFTY;
        /* find edge of minimum cost which originates at vertex v */
        for ( int j = 1; j <= n; j++ )
            if( p[ j ] == 0 && c[ v ][ j ] < costmin )
            {
                costmin = c[ v ][ j ];
                vmin = j;
            }
        *cost = *cost + costmin;
        tour[ k+1 ] = vmin;
        p[ vmin ] = 1;
        v = vmin;
    }
    tour[ n + 1 ] = i;
    *cost = *cost + c[ v ][ i ];
}
int main()
{
    int n;
    int tourCost;
    int tour[ MAXN + 1 ];
    int c[ MAXN ][ MAXN ];

    printf("\nNo. of nodes in the graph=");
    scanf("%d",&n);
    while ( getchar() != '\n' );
    for(int i = 1; i <= n; i++ )
        for(int j = 1; j <= n; j++ ) c[ i ][ j ] = INFTY;
```

```
    printf( "\nInput costs for edges with tail\nat node i and head at node j > i.");
    printf( "\nEnd input by entering 0\n");
    for(int i = 1; i <= n - 1; i++ )
    {
        for ( ; ; )
        {
            printf("Node adjacent to node %d=", i );
            int j;
            scanf( "%d", &j );
            while ( getchar() != '\n' );
            if( j != 0 )
            {
                printf( "Cost of edge (%d,%d)=", i, j );
                scanf( "%d", &c[ i ][ j ] );
                while ( getchar() != '\n' );
                c[ j ][ i ] = c[ i ][ j ];
            }
            else break;
        }
    }
    TravSMan( n, c, 1, tour, &tourCost );
    printf("\nTour cost=%d\n", tourCost );
    printf("\nTour is: ");
    for(int i = 1; i <= n + 1; i++ )
        printf("%3d",tour[i]);
    while ( getchar() != '\n' );
}
```

10.3 Laboratory Assignments

Implement modular C programs having their input and output stored in files, with names given as command line arguments, which solve the following problems using dynamic programming:

10.3.1 Mandatory Assignments

10.3.1. You are given two strings, A and B. The operations which may be performed on those strings are:

- Delete one letter from one of strings
- Insert one letter into one of strings
- Replace one of letters from one of strings with another letter

The task is to find the minimum number of operations which turn string A into string B.
I/O description. . Input: First line contains the number t of test cases. For each test cases there are two lines, with string A on the first and string B on the second. Both strings will contain only lowercase characters and they will not be longer than 2000 characters.
Output. For each test case, one line, minimum number of operations.
E.g. . Input:

```
1
food
money
```

Output:

```
4
```

10.3.2 Extra Credit Assignments

10.3.2. Annie acquired railway transport company and to stay in business she needs to lower the expenses by any means possible. There are N conductors and engineers working for her company (N is even) and $N/2$ locomotive crews need to be made. A locomotive crew consists of two people - an engineer and a conductor (his/hers assistant). An engineer must be older than his/hers assistant. Each person in an crew has a contract granting her/him two possible salaries - one as an engineer and the other as a conductor. An engineer's salary is larger than conductor's for the same person. However, it is possible that a conductor has larger salary than the engineer in the same crew. Compute the minimal amount of money Annie needs to give for the crews' salaries if she decides to spend some time to make the optimal (i.e. the cheapest) arrangement of persons in crews.
I/O description. Input. The first line of input contains integer N, $2 \leq N \leq 10,000$, N is even, the number of persons working for Annie's company. The next N lines of input contain people's salaries. The lines are sorted by

person's age, the salaries of the youngest person are given the first. Each of those N lines contains two integers separated by a space character, e and c, where $1 \leq c < e \leq 100,000$, a salary as an engineer (e) and a salary as a conductor (c).

Output. The first and only line of output should contain the minimal amount of money Annie needs to give for her employees' salaries. **E.g.** Input:

```
4
5000␣3000
6000␣2000
8000␣1000
9000␣6000
```

Output:

```
19000
```

Input:

```
6
10000␣7000
9000␣3000
6000␣4000
5000␣1000
9000␣3000
8000␣6000
```

Output:

```
32000
```

Optional Assignments

10.3.3. A target used in a shooting contest is made of concentric circles, numbered in ascending order from the outer circles to the inner circles (see figure 10.3 for an example). Each ring, bounded by two successive circles, has a strictly positive value attached to it, representing the number of points a competitor is awarded if he/she hits this ring. Find the minimum number of hits which a competitor must get in order to obtain k points. I/O description. Input: n – number of circles, k – number of points on one line. Output: integer number of shots.

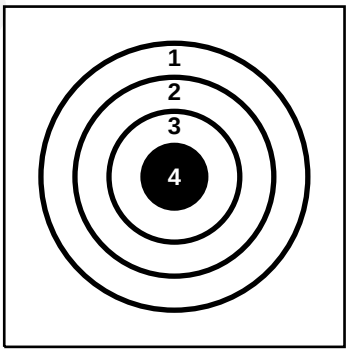


Figure 10.3: A shooting contest target.

E.g. Input

```
5 37
```

10.3.4. Two natural numbers, say A and B, and a vector v containing natural numbers are given. Show how B can be obtained from A using the values of vector v and the following operations:

a) Add to A any number of elements from v.

b) Subtract from A any number of elements from v.

Each element of v may be used for any number of times. If such solutions are possible, find the minimum number of operations which give the result B starting with the value A. I/O description. Input: A and B on one line, elements of vector v on another. Field separator is space. Output: 0 if no solution, minimum number of operations otherwise. **E.g.** Input:

```
45␣97
15␣33␣2␣31␣48␣7␣9␣21␣14␣3␣5␣1
```

10.3.5. There are n apples hanging from one branch of an apple tree. Each of the apples is at height h_i centimeters from the ground as weights g_i grams. A harvester wishes to gather a number of apples of maximum weight. But, as he rips an apple, the branch of which the apple was previously hanging becomes lighter and rises x centimeters. The harvester can pick only apples situated below a height of d centimeters. Find the maximum weight of apples which

the harvester can gather, and the order in which the apples should be selected. I/O description. Input: n, x and d on one line separated by spaces, then pairs (g_i, h_i) each on a separate line. Output maximum weight on one line, followed by the pairs describing the apples, each on one line. **E.g.** Input:

```
12 3 210
120 190
100 205
...
```

Output:

```
3250
120 190
100 205
...
```

Solve using heuristics, the following problems:

10.3.5. Find the maximum of a function $f(x)$ situated in the interval $[a, b]$.
I/O description. Input: sequence of function values, separated by one space all on one line, followed by a and b (the endpoints of the interval) on another line, eg.

```
11 22.5 -15 31.2 ...
-3 4
```

Output should be the maximum value of f on one line.

10.3.6. An undirected graph G with n vertices is given. Find the minimum number of colors needed to color all the nodes of the given graph, such that the endpoints of every edge would be of different colors. I/O description. Input: n alone on a line, followed by vertex pairs – one pair on a line, e.g.

```
9
1 2
2 5
5 7
...
```

Output: minimum number of colors, c, on one line followed by vertex pairs with colors, i.e. (v_i, c_j), where v_i, $1 \leq i \leq n$ is vertex number and c_j $1 \leq j \leq c$ is color number, e.g.

```
3
1 1
2 3
5 1
7 2
...
```

10.3.7. Given a $n-$vertex undirected graph $G = (V, E)$, find a maximal number of vertices, $V' \subset V$ made of nodes *not* connected by an edge. I/O description. Input: same as the one of the previous problem. Output: list of vertex numbers, all on one line, separated by one space.

10.4 Supplementary example implementations

10.4.1 Example 1. Optimal Binary Search Trees

Listing 10.4 shown an implementation of the optimal binary search tree problem with string keys.

Listing 10.4: Solution to the optimal binary search tree problem. Here, keys are character strings of length at most 40

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

#define NMAX 21
#define MAX_KEYLEN 41

typedef struct OptimalBST_T
{
 char *key;
```

```
 struct OptimalBST_T *left, *right;
} OptimalBST_T;

float costs[NMAX][NMAX]; //cost matrix
float weights[NMAX][NMAX]; //weight matrix
int roots[NMAX][NMAX]; //root matrix
float q[NMAX]; //unsuccesful searches
float p[NMAX]; //frequencies
int numberOfKeys; //number of keys in the tree
char keys[NMAX][MAX_KEYLEN];

void computeCostAndRoots()
{
//Construct weight matrix weights
 for(int i = 0; i <= numberOfKeys; i++)
 {
  weights[i][i] = q[i];
  for(int j = i + 1; j <= numberOfKeys; j++)
   weights[i][j] = weights[i][j-1] + p[j] + q[j];
 }
//Construct cost matrix costs and root matrix roots
 for(int i = 0; i <= numberOfKeys; i++)
  costs[i][i] = weights[i][i];
 for(int i = 0; i <= numberOfKeys - 1; i++)
 {
  int j = i + 1;
  costs[i][j] = costs[i][i] + costs[j][j] + weights[i][j];
  roots[i][j] = j;
 }
 for(int h = 2; h <= numberOfKeys; h++)
  for(int i = 0; i <= numberOfKeys - h; i++)
  {
   int j = i + h;
   int m = roots[i][j-1];
   float minCost = costs[i][m-1] + costs[m][j];
   for(int k = m+1; k <= roots[i+1][j]; k++)
   {
    float newCost = costs[i][k-1] + costs[k][j];
    if(newCost < minCost)
    {
     m = k;
     minCost = newCost;
    }
   }
   costs[i][j] = weights[i][j] + minCost;
   roots[i][j] = m;
  }

//Display weight matrix weights
 printf("\nThe weight matrix weights:\n");
 for(int i = 0; i <= numberOfKeys; i++)
 {
  for(int j = i; j <= numberOfKeys; j++)
   printf("%f ", weights[i][j]);
  printf("\n");
 }
//Display Cost matrix costs
 printf("\nThe cost matrix costs:\n");
 for(int i = 0; i <= numberOfKeys; i++)
 {
  for(int j = i; j <= numberOfKeys; j++)
   printf("%f ", costs[i][j]);
  printf("\n");
 }
//Display root matrix roots
 printf("\nThe root matrix roots:\n");
 for(int i = 0; i <= numberOfKeys; i++)
 {
  for(int j = i; j <= numberOfKeys; j++)
   printf("%d ", roots[i][j]);
  printf("\n");
 }
}
//Construct the optimal binary search tree
```

```
OptimalBST_T *buildTree(int i, int j)
{
 OptimalBST_T *p;
 if(i == j)
  p = NULL;
 else
 {
  p =(OptimalBST_T *) malloc(sizeof(OptimalBST_T));
  if (p)
  {
   p->key = keys[roots[i][j]];
   p->left = buildTree(i, roots[i][j] - 1); //left subtree
   p->right = buildTree(roots[i][j], j); //right subtree
  }
  else
  {
   printf("No core in buildTree\n");
   exit(3);
  }
 }
 return p;
}
// returns index of key in p
int findProbability(const char *key)
{
 for (int i = 1; i <= numberOfKeys; i++)
 {
  if (strcmp(keys[i], key) == 0)
  return i;
 }
 return 0;
}
//Display the optimal binary search tree
void displayTree(OptimalBST_T *root, int level)
{
 if(root != NULL)
 {
  if (root->right == NULL)
  {
   for(int i = 0; i <= level + 1; i++)
    printf("   ");
   printf(">%s (q=%.2f)\n", root->key, q[findProbability(root->key)]);
  }
  displayTree(root->right, level + 1);
  for(int i = 0; i <= level; i++)
   printf("   ");
  printf("%s (d=%d, p=%.2f)\n", root->key, level, p[findProbability(root->key)]);
  if (root->left == NULL)
  {
   for(int i = 0; i <= level + 1; i++)
    printf("   ");
   printf("<%s (q=%.2f)\n", root->key, q[findProbability(root->key)-1]);
  }
  displayTree(root->left, level + 1);
 }
}
// build an optimal binary search tree
OptimalBST_T *buildOptimalBST()
{
 computeCostAndRoots();
 printf("costs[0] = %f weights[0] = %f\n", costs[0][numberOfKeys],
     weights[0][numberOfKeys]);
 float expectedSearchCost =
  costs[0][numberOfKeys]/weights[0][numberOfKeys];
 printf("Expected search cost is: %f\n", expectedSearchCost);
 return buildTree(0, numberOfKeys);
}
int main(int argc, char *argv[])
{
 const char *explain[]=
 {
  "Builds an optimal binary search tree with keys provided in input file\n"
  "Input file structure:\n",
  "\tLine 1: number of keys, n\n",
```

```
  "\tn lines with 2 values separated by 1 space: key value (string) occurence frequency(integer)\n"
      ,
  "\tn+1 lines of 1 integer each: occurence frequencies of unsuccessful searches\n",
  ""
 };
 if (argc < 2)
 {
  printf("Usage:\n\t%s inputFilename\n", strrchr(argv[0], '\\')+1);
  for (int i=0; explain[i][0]; i++)
   printf("%s", explain[i]);
  exit(1);
 }
 FILE *fp = fopen(argv[1], "r");
 if (fp == NULL)
 {
  perror(argv[1]);
  exit(2);
 }
 fscanf(fp, "%d", &numberOfKeys);
 if (numberOfKeys < 1 || numberOfKeys > NMAX-1)
 {
  printf("Number of keys %d not in admissible range [1..%d]\n", numberOfKeys, NMAX-1);
  exit(4);
 }
 for(int i = 1; i <= numberOfKeys; i++)
 {
  fscanf(fp, "%s %f", keys[i], &p[i]);
 }
 for(int i = 0; i <= numberOfKeys; i++)
 {
  fscanf(fp, "%f",&q[i]);
 }
 OptimalBST_T *root = buildOptimalBST();
 displayTree(root, 0);
 return 0;
}
```

10.4.2 Example 2. Longest Common Subsequence

The example shown in Listing 10.5 is an implementation for the longest common subsequence problem. Figure 10.4.2 show a trace of the algorithm for sequences "ADBCDBAB", and "ABDCBABC".

Listing 10.5: Solution to the the longest common subsequence problem.

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

#define SKIPX 1
#define SKIPY 2
#define ADDXY 3

// allocate a matrix of integers
int **allocMatrix(int rows, int cols)
{
 int **matrix = (int **)malloc( (rows + 1) * sizeof(int) );
 if (matrix)
 {
  matrix[0] = (int*) malloc( (rows + 1) * (cols + 1) * sizeof(int) );
  for(int i = 1; i <= rows; ++i)
  {
   matrix[i] = matrix[i - 1] + cols + 1;
  }
 }
 return matrix;
}

char* LCS(char* x, char* y)
{
 int m = strlen(x);
 int n = strlen(y);

 int** cost = allocMatrix(m + 1, n + 1);
```

```
int** back = allocMatrix(m + 1, n + 1);
// initialize column zero
for(int i = 0; i <= m; ++i)
{
 cost[i][0] = 0;
 back[i][0] = SKIPX;
}
// initialize row zero
for(int j = 0; j <= n; ++j)
{
 cost[0][j] = 0;
 back[0][j] = SKIPY;
}
for(int i = 1; i <= m; ++i)
{
 for(int j = 1; j <= n; ++j)
 {
  if( x[i - 1] == y[j - 1] )
  {   // current character match
   cost[i][j] = cost[i - 1][j - 1] + 1;
   back[i][j] = ADDXY;
  }
  else if( cost[i - 1][j] > cost[i][j - 1] )
  {   // skip char from first string
   cost[i][j] = cost[i - 1][j];
   back[i][j] = SKIPX;
  }
  else
  {   // skip char from second string
   cost[i][j] = cost[i][j - 1];
   back[i][j] = SKIPY;
  }
 }
}
//Display Cost matrix costs
printf("\nThe cost matrix costs:\n");
for(int i = 0; i <= m; i++)
{
 for(int j = i; j <= n; j++)
  printf("%d ", cost[i][j]);
 printf("\n");
}
       //Display root matrix roots
printf("\nThe root matrix roots:\n");
for(int i = 0; i <= m; i++)
{
 for(int j = i; j <= n; j++)
  printf("%d ", back[i][j]);
 printf("\n");
}

/* The length of the longest substring is cost[n][m] */
int pos = cost[m][n];
char *lcs = (char *) malloc( (pos + 1) * sizeof(char) );
lcs[pos--] = '\0';
// Trace the backtrack matrix.
int i = m;
int j = n;
while( i > 0 || j > 0 )
{
 if( back[i][j] == ADDXY )
 {
  lcs[pos--] = x[i - 1];
  i--;
  j--;
 }
 else if( back[i][j] == SKIPX )
  i--;
 else
  j--; // SKIPY
}
free(cost[0]);
free(back[0]);
free(cost);
```

```
 free(back);
 return lcs;
}

int main(int argc, char* argv[])
{
 if (argc < 3)
 {
  printf("Must provide 2 strings as arguments\n");
  exit(1);
 }
 printf("LCS of %s and %s is: %s\n", argv[1], argv[2], LCS(argv[1],argv[2]));
}
```

		Y:	0	1	2	3	4	5	6	7	8	$= n$
				A	B	D	C	B	A	B	C	
0			0	0	0	0	0	0	0	0	0	
1	A		0									
2	D		0									
3	B		0									
X: 4	C		0									
5	D		0									
6	B		0									
7	A		0									
$m = 8$	B		0									

		Y:	0	1	2	3	4	5	6	7	8	$= n$
				A	B	D	C	B	A	B	C	
0			0	0	0	0	0	0	0	0	0	
				↘	→	→	→	→	↘	→	→	
1	A		0	1	1	1	1	1	1	1	1	
2	D		0									
3	B		0									
X: 4	C		0									
5	D		0									
6	B		0									
7	A		0									
$m = 8$	B		0									

		Y:	0	1	2	3	4	5	6	7	8	$= n$
				A	B	D	C	B	A	B	C	
0			0	0	0	0	0	0	0	0	0	
				↘	→	→	→	→	↘	→	→	
1	A		0	1	1	1	1	1	1	1	1	
				↓	↓	↘	→	→	→	→	→	
2	D		0	1	1	2	2	2	2	2	2	
3	B		0									
X: 4	C		0									
5	D		0									
6	B		0									
7	A		0									
$m = 8$	B		0									

		Y:	0	1	2	3	4	5	6	7	8	$= n$
				A	B	D	C	B	A	B	C	
0			0	0	0	0	0	0	0	0	0	
				↘	→	→	→	→	↘	→	→	
1	A		0	1	1	1	1	1	1	1	1	
				↓	↓	↘	→	→	→	→	→	
2	D		0	1	1	2	2	2	2	2	2	
				↓	↘	↓	↓	↘	→	↘	→	
3	B		0	1	2	2	2	3	3	3	3	
X: 4	C		0									
5	D		0									
6	B		0									
7	A		0									
$m = 8$	B		0									

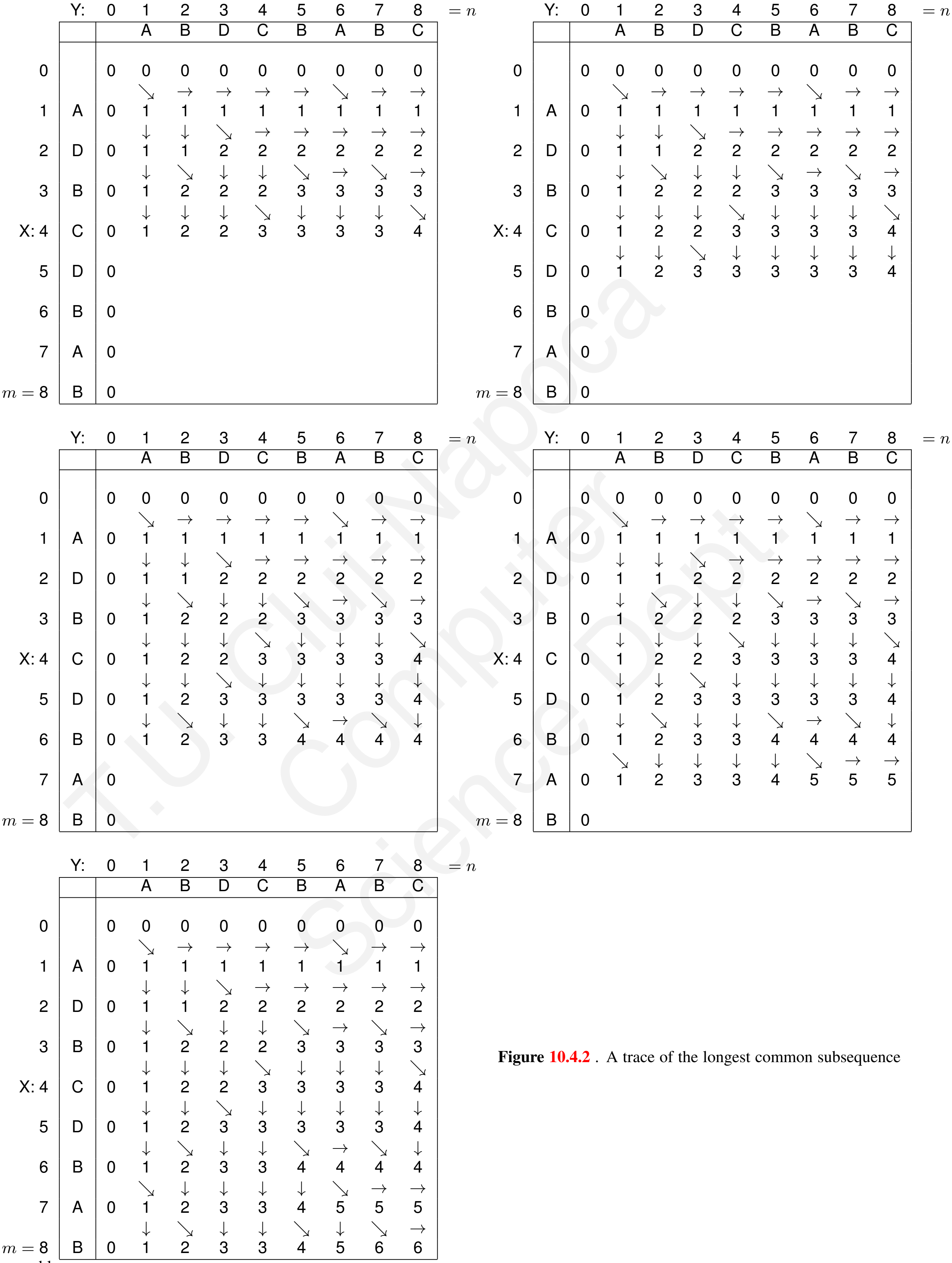


Figure 10.4.2 . A trace of the longest common subsequence problem.