LAB ASSESSMENT 2

Name: Karan Sehgal

Registration Number: 22BCE3939

Course Name: Design and Analysis of Algorithms

Course Code: BCSE204P

Submitted to: : Prof. Gayatri P

Question 1:

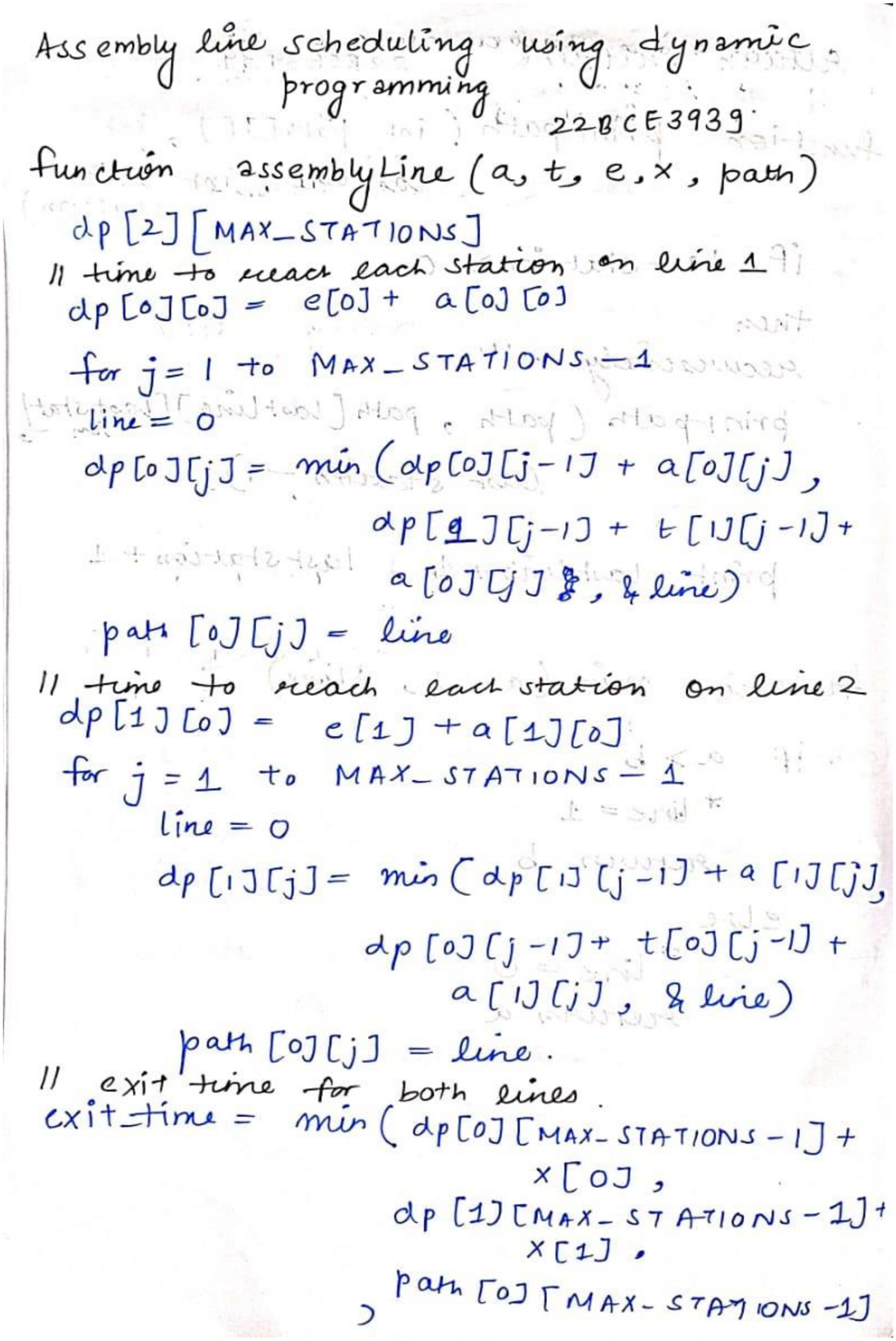
Assembly line scheduling using DP approach.

Problem Statement:

You are given two assembly lines, each with n stations. Each station takes a certain amount of time to process a unit and each assembly line has an entry and exit point. The units move through the stations in a sequential manner, and there is a transfer time associated with moving a unit from one station to the next.

Your goal is to find the most efficient way to schedule the units through the assembly lines to minimize the total time required for the entire process.

Pseudo-Code:



Assembly line scheduling using dynamic programming

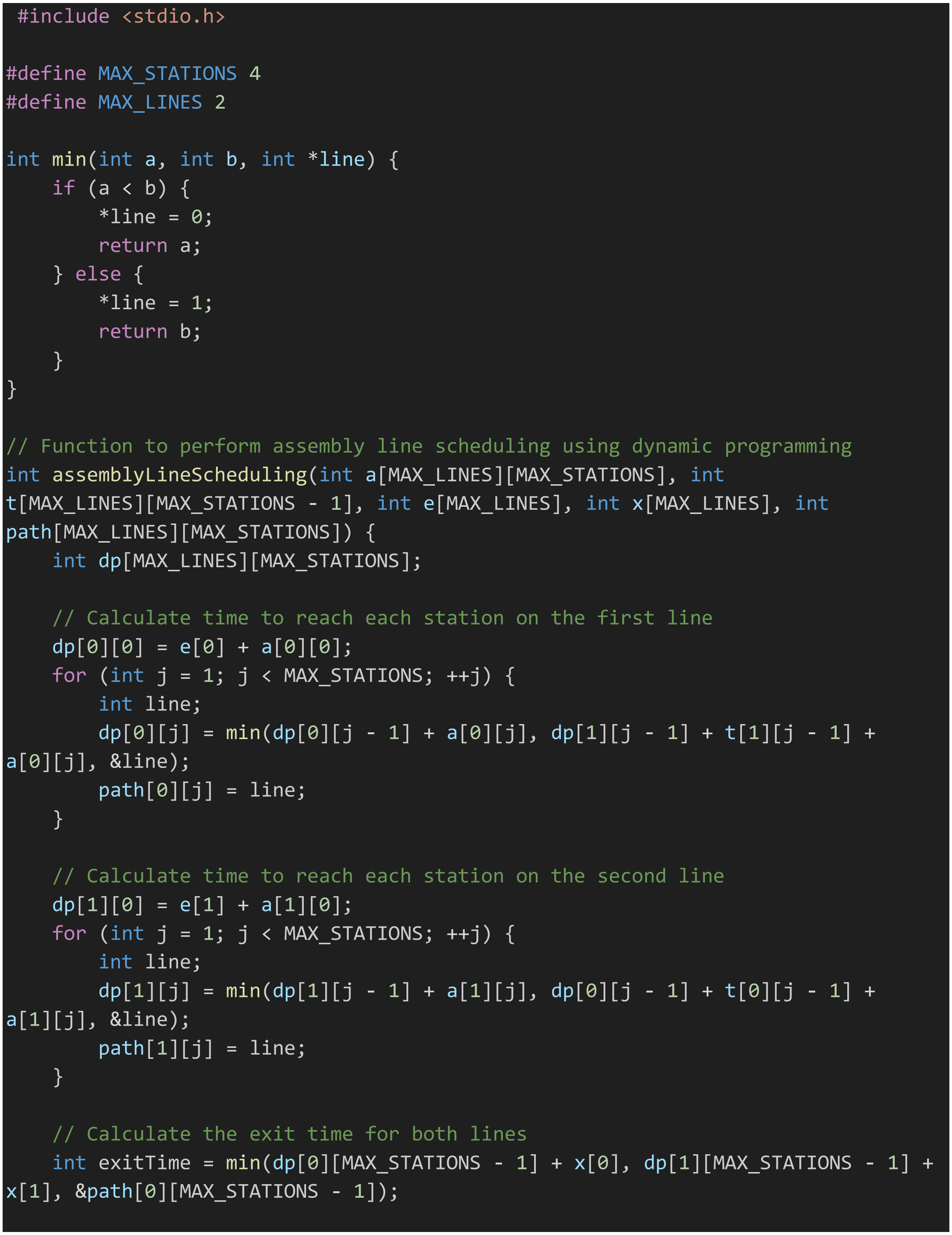
22BCE3939

```
function assemblyLine (a, t, e, x, path)
  dp [2] [MAX_STATIONS]
  // time to reach each station on line 1
  dp [0] [0] = e[0] + a [0] [0]

  for j = 1 to MAX_STATIONS - 1
    line = 0
    dp [0] [j] = min (dp [0] [j-1] + a [0] [j],
                      dp [1] [j-1] + t [1] [j-1] +
                      a [0] [j] ~~&~~, & line)
    path [0] [j] = line
  // time to reach each station on line 2
  dp [1] [0] = e [1] + a [1] [0]
  for j = 1 to MAX_STATIONS - 1
      line = 0
      dp [1] [j] = min ( dp [1] [j-1] + a [1] [j],
                         dp [0] [j-1] + t [0] [j-1] +
                         a [1] [j] , & line)
      path [0] [j] = line.
  // exit time for both lines
  exit_time = min ( dp [0] [MAX_STATIONS - 1] +
                    x [0] ,
                    dp [1] [MAX_STATIONS - 1] +
                    x [1] ,
                    path [0] [MAX_STATIONS -1]
                  )
```

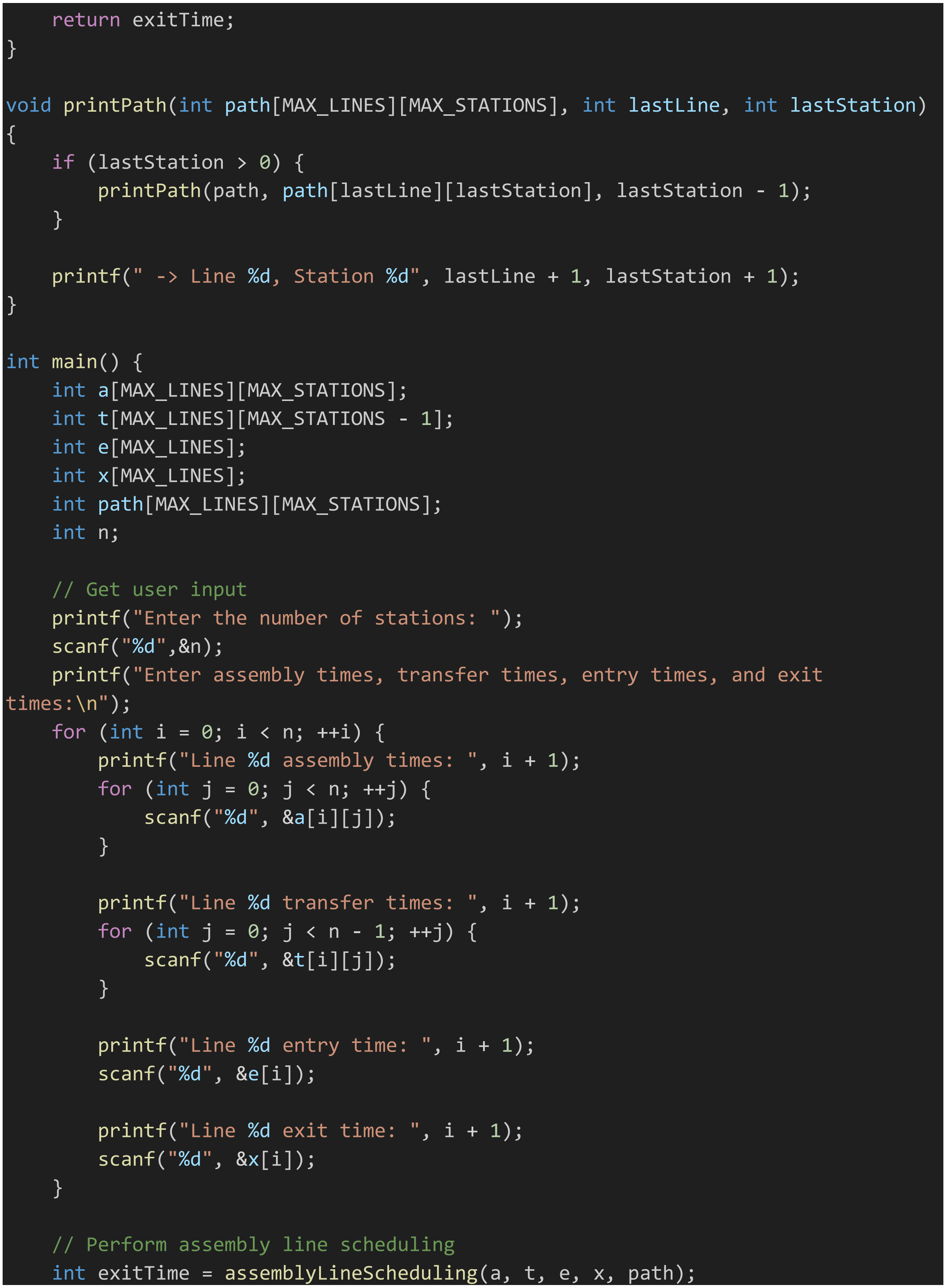
```
return exittime        22BCE3939

function printpath ( int path[][] , int
                     lastline , int last
                     station)
    if last station > 0
    then
    recursively call
    printpath ( path , path [lastline][laststation],
                last station - 1);

    print lastline + 1 , laststation + 1

function min (a , b , *line)
    if a > b
        *line = 1
        return b
    else
        *line = 0
        return a
```

```
function main ()   //Driver Code
                                  Max _lines = 2
// Get user input
                                       22BCE3939
Read no. of . stations as n

for i = 0 to MAX_LINES-1
    for j = 0 to n
        read a[i][j]

    for j = 0 to n - 2
        read t[i][j]

    read e[i]

    read x[i]

exittime = assemblyLine (a, t, e, x, path)

print "Minimum Time to Exit" + exit-time
print "Optimal path:"
call
printpath (path, path[0][MAX_STATIONS-1],
                                   n - 1)
```

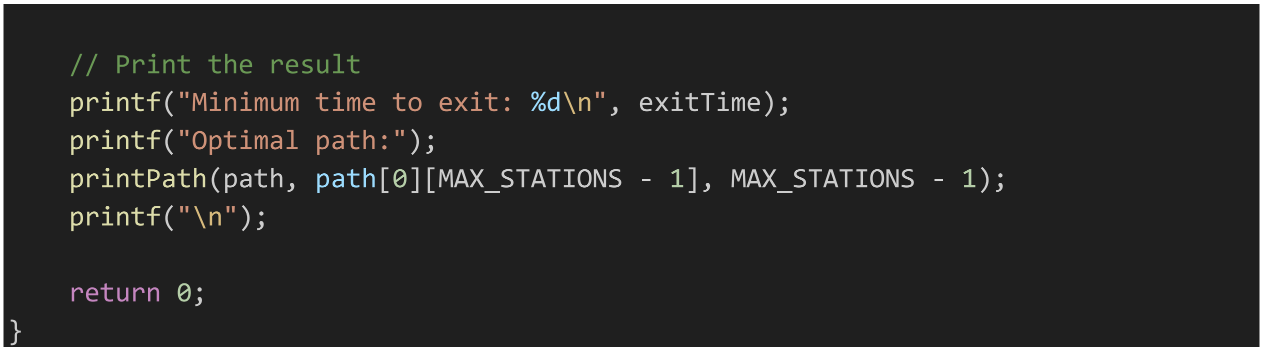
Source code:



```
 #include <stdio.h>

#define MAX_STATIONS 4
#define MAX_LINES 2

int min(int a, int b, int *line) {
    if (a < b) {
        *line = 0;
        return a;
    } else {
        *line = 1;
        return b;
    }
}

// Function to perform assembly line scheduling using dynamic programming
int assemblyLineScheduling(int a[MAX_LINES][MAX_STATIONS], int
t[MAX_LINES][MAX_STATIONS - 1], int e[MAX_LINES], int x[MAX_LINES], int
path[MAX_LINES][MAX_STATIONS]) {
    int dp[MAX_LINES][MAX_STATIONS];

    // Calculate time to reach each station on the first line
    dp[0][0] = e[0] + a[0][0];
    for (int j = 1; j < MAX_STATIONS; ++j) {
        int line;
        dp[0][j] = min(dp[0][j - 1] + a[0][j], dp[1][j - 1] + t[1][j - 1] +
a[0][j], &line);
        path[0][j] = line;
    }

    // Calculate time to reach each station on the second line
    dp[1][0] = e[1] + a[1][0];
    for (int j = 1; j < MAX_STATIONS; ++j) {
        int line;
        dp[1][j] = min(dp[1][j - 1] + a[1][j], dp[0][j - 1] + t[0][j - 1] +
a[1][j], &line);
        path[1][j] = line;
    }

    // Calculate the exit time for both lines
    int exitTime = min(dp[0][MAX_STATIONS - 1] + x[0], dp[1][MAX_STATIONS - 1] +
x[1], &path[0][MAX_STATIONS - 1]);
```

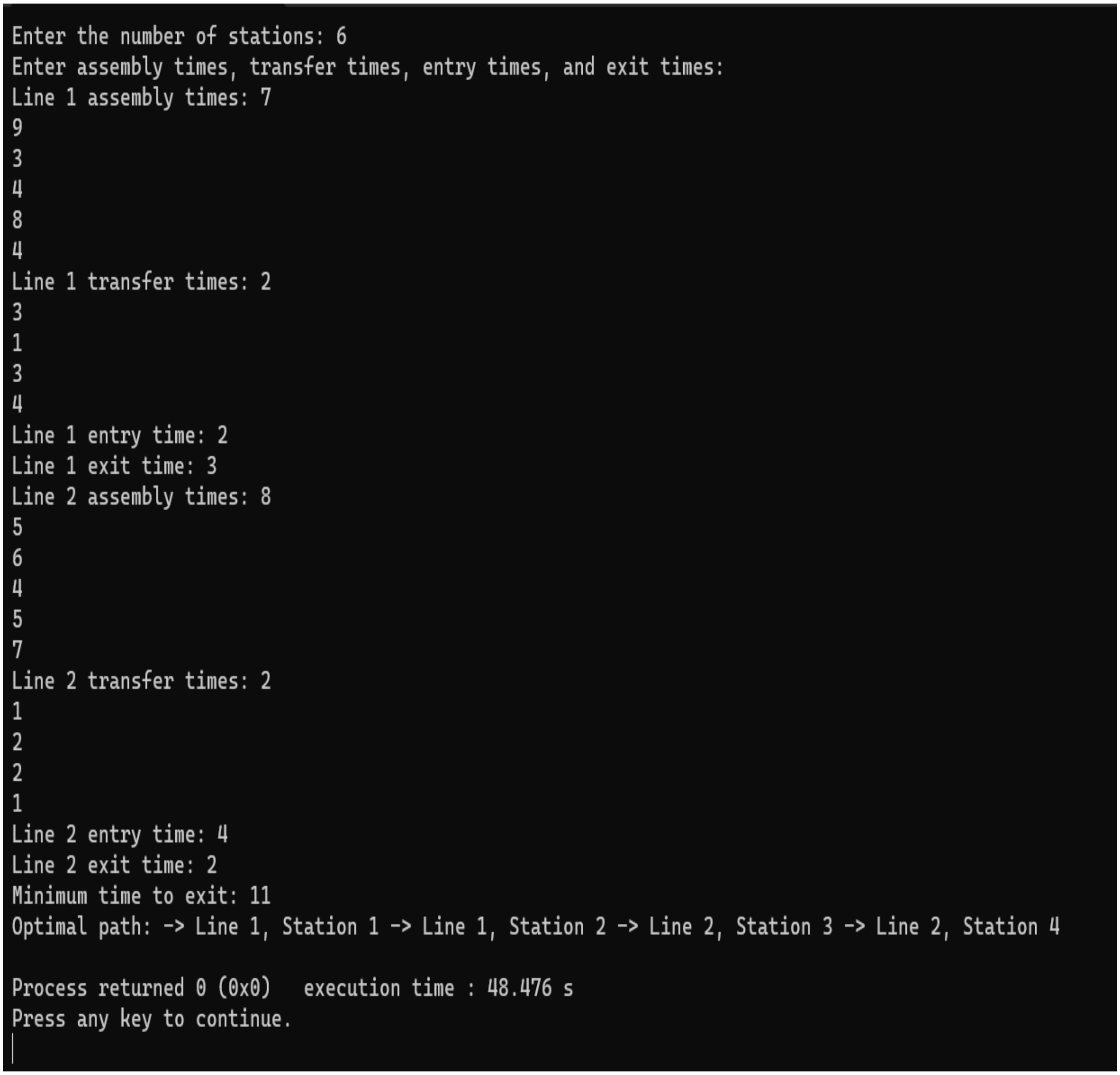


```
    return exitTime;
}

void printPath(int path[MAX_LINES][MAX_STATIONS], int lastLine, int lastStation)
{
    if (lastStation > 0) {
        printPath(path, path[lastLine][lastStation], lastStation - 1);
    }

    printf(" -> Line %d, Station %d", lastLine + 1, lastStation + 1);
}

int main() {
    int a[MAX_LINES][MAX_STATIONS];
    int t[MAX_LINES][MAX_STATIONS - 1];
    int e[MAX_LINES];
    int x[MAX_LINES];
    int path[MAX_LINES][MAX_STATIONS];
    int n;

    // Get user input
    printf("Enter the number of stations: ");
    scanf("%d",&n);
    printf("Enter assembly times, transfer times, entry times, and exit
times:\n");
    for (int i = 0; i < n; ++i) {
        printf("Line %d assembly times: ", i + 1);
        for (int j = 0; j < n; ++j) {
            scanf("%d", &a[i][j]);
        }

        printf("Line %d transfer times: ", i + 1);
        for (int j = 0; j < n - 1; ++j) {
            scanf("%d", &t[i][j]);
        }

        printf("Line %d entry time: ", i + 1);
        scanf("%d", &e[i]);

        printf("Line %d exit time: ", i + 1);
        scanf("%d", &x[i]);
    }

    // Perform assembly line scheduling
    int exitTime = assemblyLineScheduling(a, t, e, x, path);
```

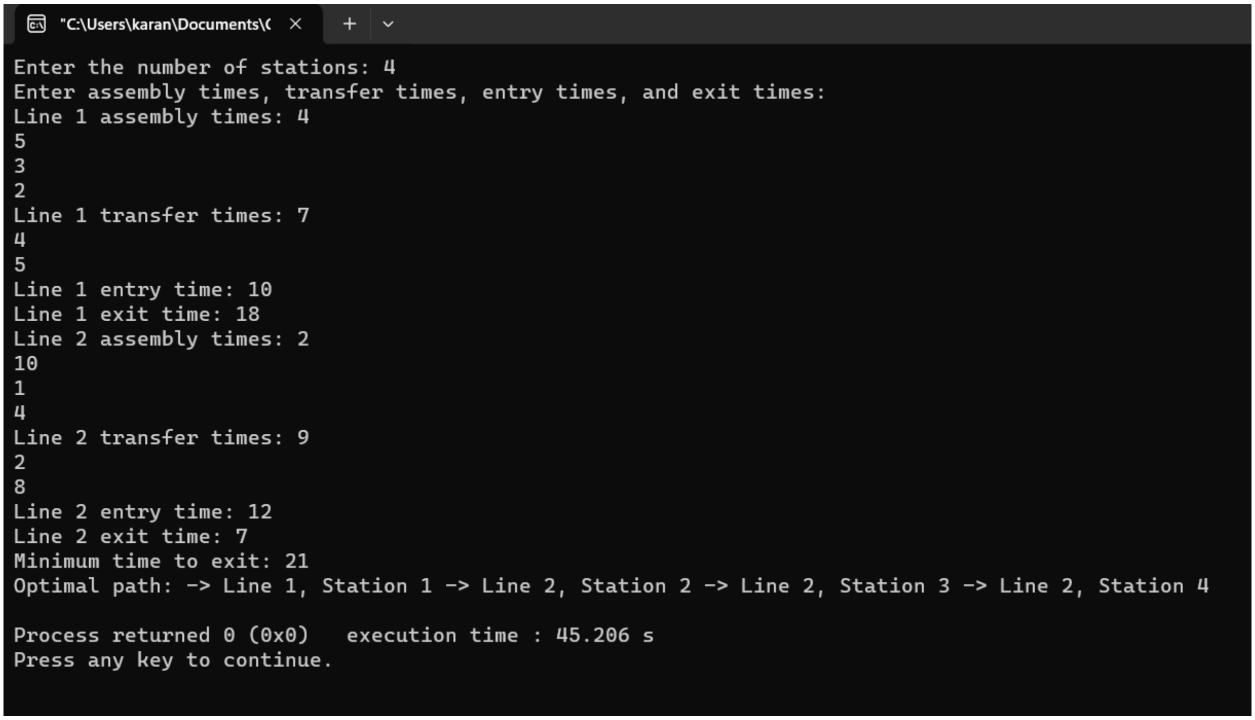


```c
    // Print the result
    printf("Minimum time to exit: %d\n", exitTime);
    printf("Optimal path:");
    printPath(path, path[0][MAX_STATIONS - 1], MAX_STATIONS - 1);
    printf("\n");

    return 0;
}
```

Output :



```
Enter the number of stations: 6
Enter assembly times, transfer times, entry times, and exit times:
Line 1 assembly times: 7
9
3
4
8
4
Line 1 transfer times: 2
3
1
3
4
Line 1 entry time: 2
Line 1 exit time: 3
Line 2 assembly times: 8
5
6
4
5
7
Line 2 transfer times: 2
1
2
2
1
Line 2 entry time: 4
Line 2 exit time: 2
Minimum time to exit: 11
Optimal path: -> Line 1, Station 1 -> Line 1, Station 2 -> Line 2, Station 3 -> Line 2, Station 4

Process returned 0 (0x0)   execution time : 48.476 s
Press any key to continue.
```

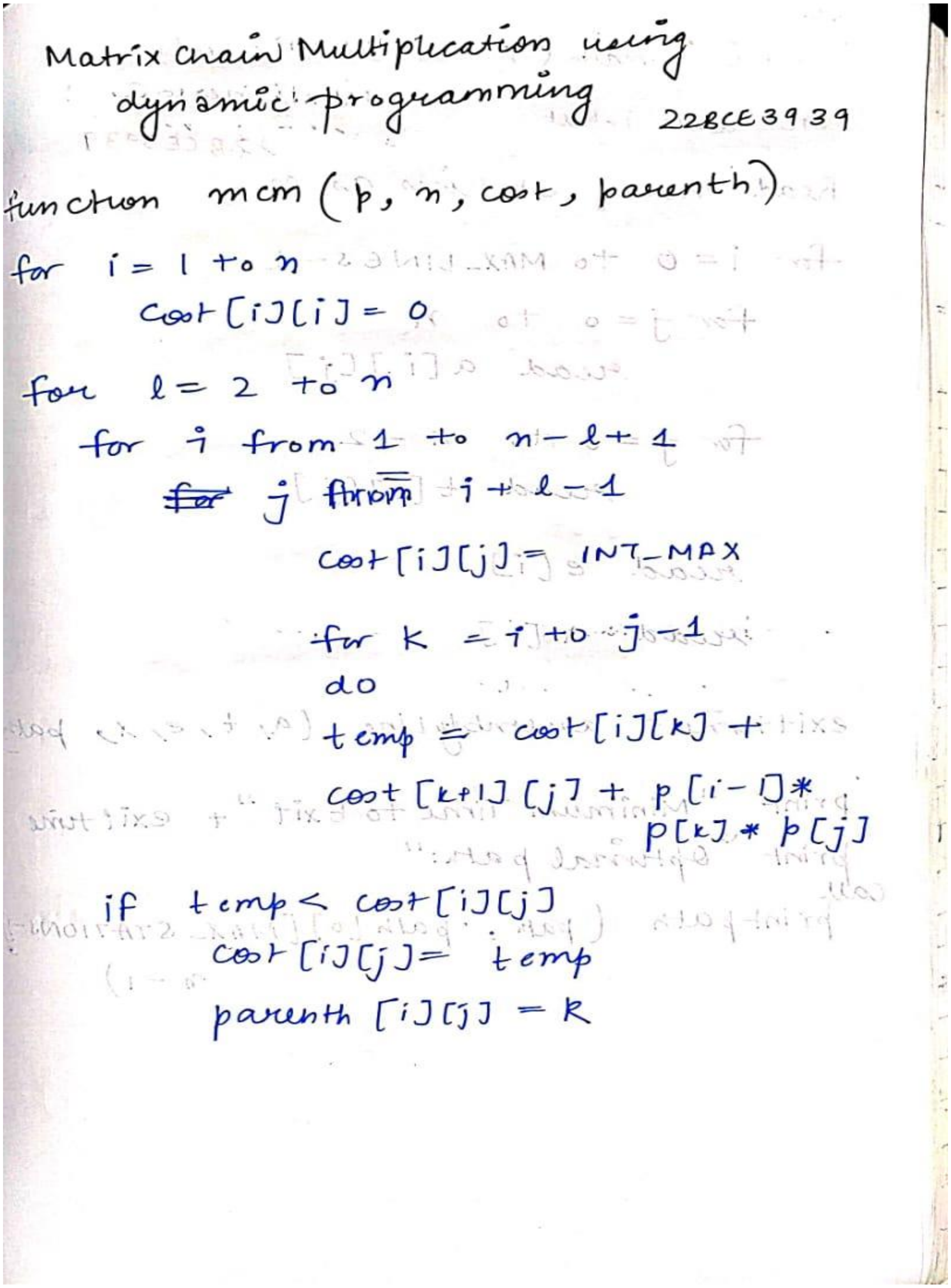


```
Enter the number of stations: 4
Enter assembly times, transfer times, entry times, and exit times:
Line 1 assembly times: 4
5
3
2
Line 1 transfer times: 7
4
5
Line 1 entry time: 10
Line 1 exit time: 18
Line 2 assembly times: 2
10
1
4
Line 2 transfer times: 9
2
8
Line 2 entry time: 12
Line 2 exit time: 7
Minimum time to exit: 21
Optimal path: -> Line 1, Station 1 -> Line 2, Station 2 -> Line 2, Station 3 -> Line 2, Station 4

Process returned 0 (0x0)   execution time : 45.206 s
Press any key to continue.
```

Question 2:

Matrix chain multiplication using DP approach

Pseudo-Code:



Matrix chain Multiplication using dynamic programming

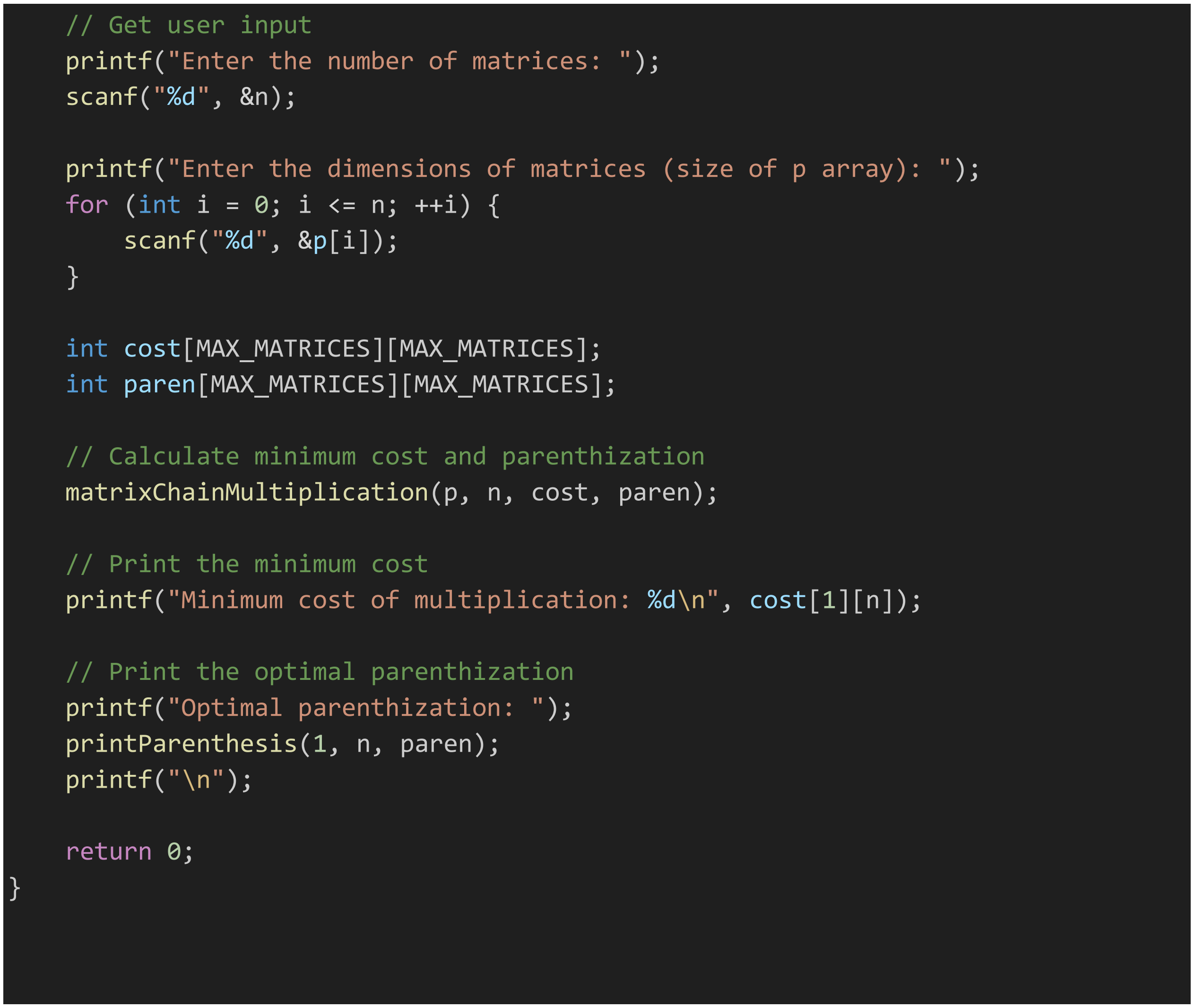
22BCE3939

```
function mcm (p, n, cost, parenth)
for i = 1 to n
      Cost [i][i] = 0

for l = 2 to n
   for i from 1 to n - l + 1
         ~~for~~ j ~~from~~ = i + l - 1
              cost [i][j] = INT_MAX

              for k = i to j - 1
              do
              temp = cost [i][k] +
              cost [k+1] [j] + p [i - 1]*
                                   p [k] * p [j]

   if temp < cost [i][j]
        cost [i][j] = temp
        parenth [i][j] = k
```

```
function
print parethesis (i, j, parenth)
                            // for printing
                            order of multiplicat

if i is equal to j                    22BCE3939
    print "A" + i
else
    print "("
    printparenthesis (i, parenth[i][j], paren
    print parenthesis (parenth [i][j] + 1, j,
                        parenth)
    print ")"
function main(): // Driver Code
// Get user input
read n
for i from 0 to n
    read p[i]

call mcm (p, n, cost, parenth)

print "Min cost:" + cost [1][n]

print "Optimial Parenthization: "
printparenthesis (1, n, parenth)
```

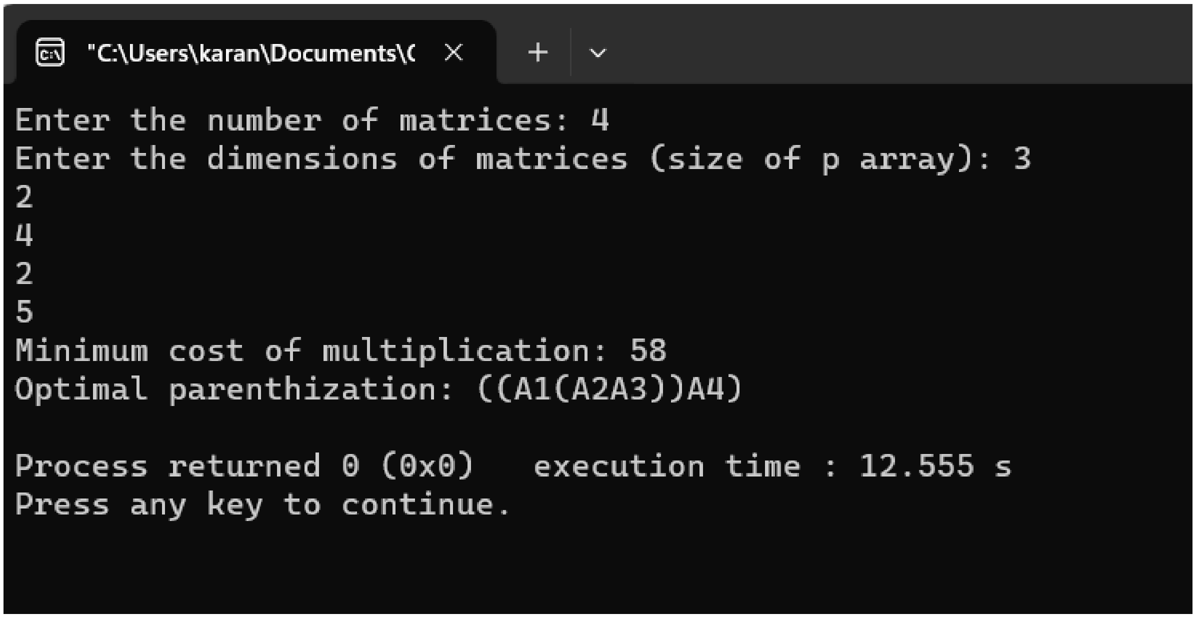
Source Code:

```c
#include <stdio.h>

#define MAX_MATRICES 100

void matrixChainMultiplication(int p[], int n, int
cost[MAX_MATRICES][MAX_MATRICES], int paren[MAX_MATRICES][MAX_MATRICES]) {
    for (int i = 1; i <= n; ++i) {
        cost[i][i] = 0;
    }

    for (int len = 2; len <= n; ++len) {
        for (int i = 1; i <= n - len + 1; ++i) {
            int j = i + len - 1;
            cost[i][j] = __INT_MAX__;

            for (int k = i; k < j; ++k) {
                int tempCost = cost[i][k] + cost[k + 1][j] + p[i - 1] * p[k] *
p[j];

                if (tempCost < cost[i][j]) {
                    cost[i][j] = tempCost;
                    paren[i][j] = k;
                }
            }
        }
    }
}

void printParenthesis(int i, int j, int paren[MAX_MATRICES][MAX_MATRICES]) {
    if (i == j) {
        printf("A%d", i);
    } else {
        printf("(");
        printParenthesis(i, paren[i][j], paren);
        printParenthesis(paren[i][j] + 1, j, paren);
        printf(")");
    }
}

int main() {
    int n;
    int p[MAX_MATRICES];
```

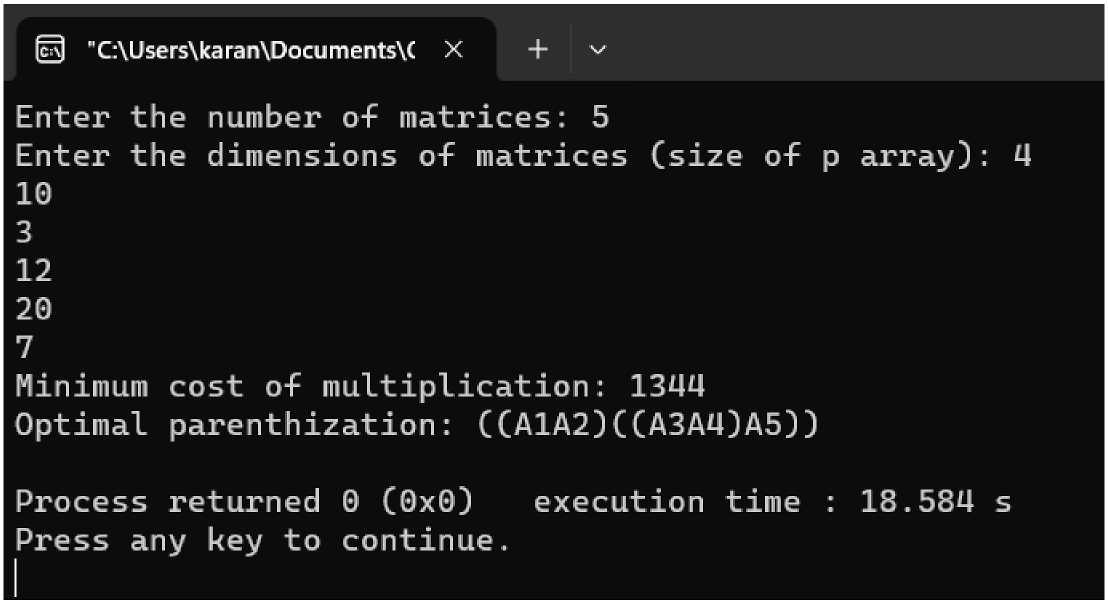


```c
    // Get user input
    printf("Enter the number of matrices: ");
    scanf("%d", &n);

    printf("Enter the dimensions of matrices (size of p array): ");
    for (int i = 0; i <= n; ++i) {
        scanf("%d", &p[i]);
    }

    int cost[MAX_MATRICES][MAX_MATRICES];
    int paren[MAX_MATRICES][MAX_MATRICES];

    // Calculate minimum cost and parenthization
    matrixChainMultiplication(p, n, cost, paren);

    // Print the minimum cost
    printf("Minimum cost of multiplication: %d\n", cost[1][n]);

    // Print the optimal parenthization
    printf("Optimal parenthization: ");
    printParenthesis(1, n, paren);
    printf("\n");

    return 0;
}
```

Output:



```
Enter the number of matrices: 4
Enter the dimensions of matrices (size of p array): 3
2
4
2
5
Minimum cost of multiplication: 58
Optimal parenthization: ((A1(A2A3))A4)

Process returned 0 (0x0)   execution time : 12.555 s
Press any key to continue.
```



```
Enter the number of matrices: 5
Enter the dimensions of matrices (size of p array): 4
10
3
12
20
7
Minimum cost of multiplication: 1344
Optimal parenthization: ((A1A2)((A3A4)A5))

Process returned 0 (0x0)   execution time : 18.584 s
Press any key to continue.
```

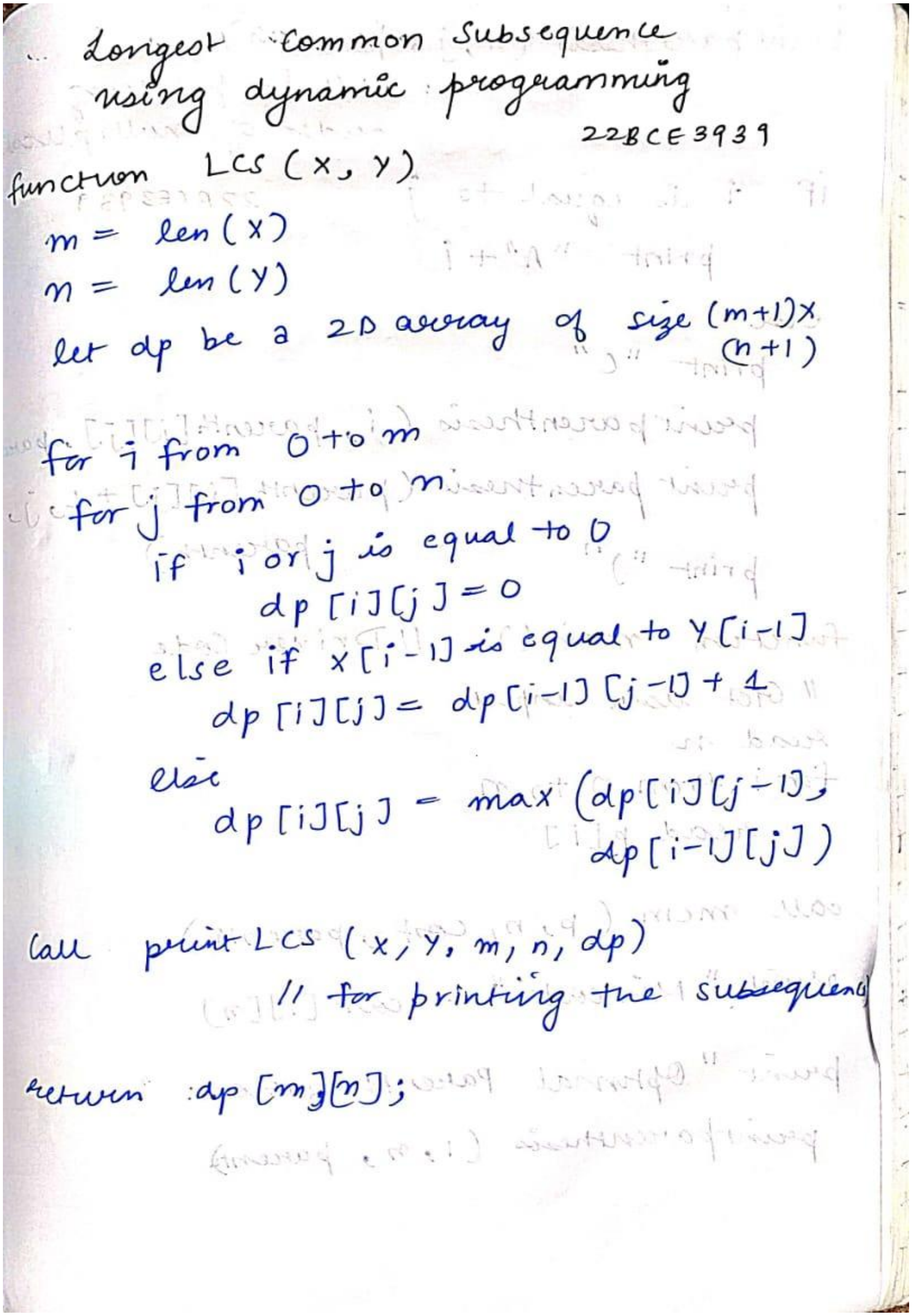
Question 3:

Longest common subsequence using DP approach.

Problem Statement:

You are given two sequences of characters, and your task is to find the length of the Longest Common Subsequence (LCS) of these sequences. A subsequence is a sequence that appears in the same relative order but not necessarily contiguous. For example, "abc", "abg", "bdf", "aeg", "acefg", … are subsequences of "abcdefg"

Pseudocode:



Longest Common Subsequence
using dynamic programming

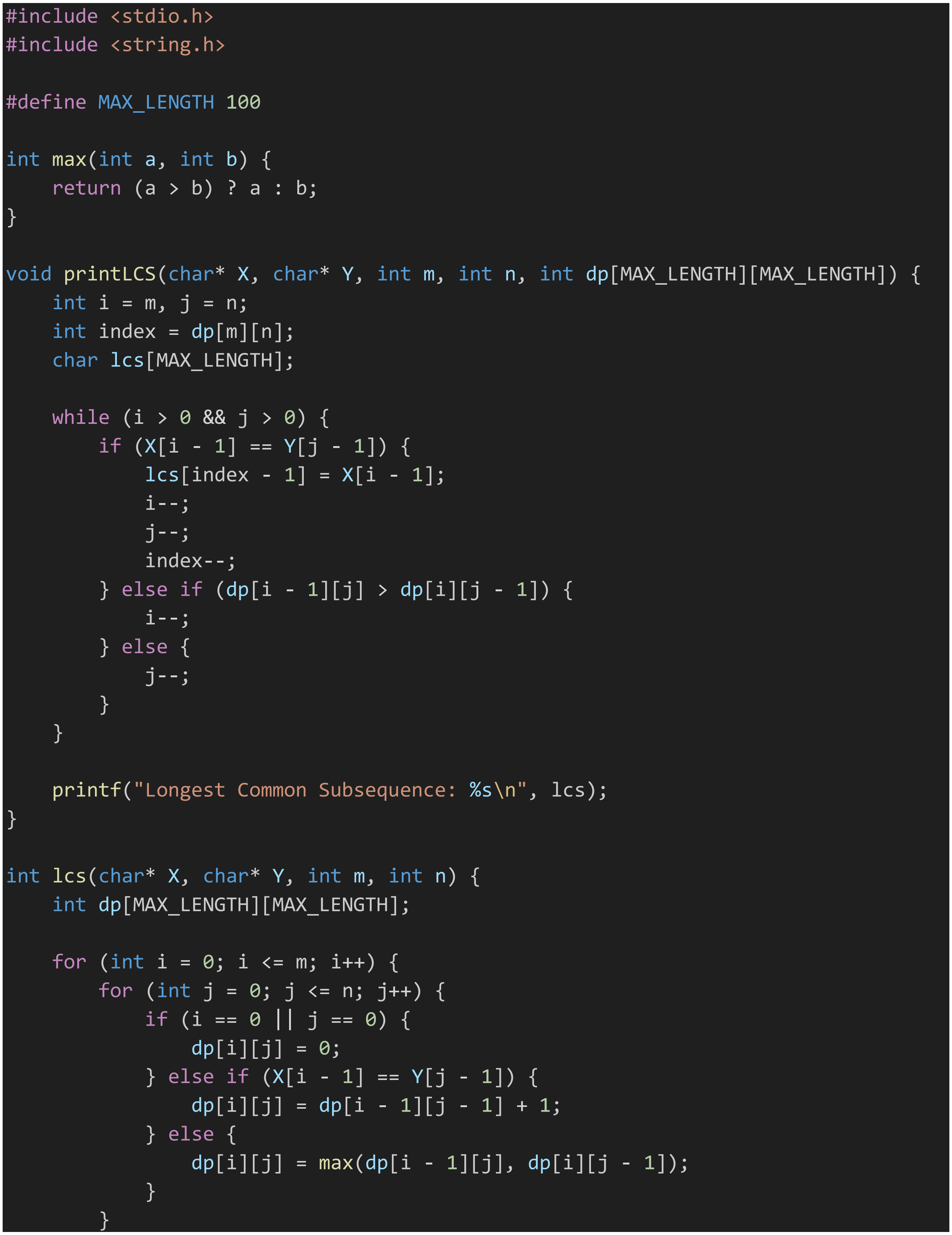
22BCE3939

```
function LCS (x, y)
  m = len(x)
  n = len(y)
  let dp be a 2D array of size (m+1)x
                                 (n+1)

  for i from 0 to m
    for j from 0 to n
      if i or j is equal to 0
        dp[i][j] = 0
      else if x[i-1] is equal to y[i-1]
        dp[i][j] = dp[i-1][j-1] + 1

      else
        dp[i][j] = max (dp[i][j-1],
                        dp[i-1][j])

  call print LCS (x, y, m, n, dp)
                 // for printing the subsequenc

  return dp[m][n];
```

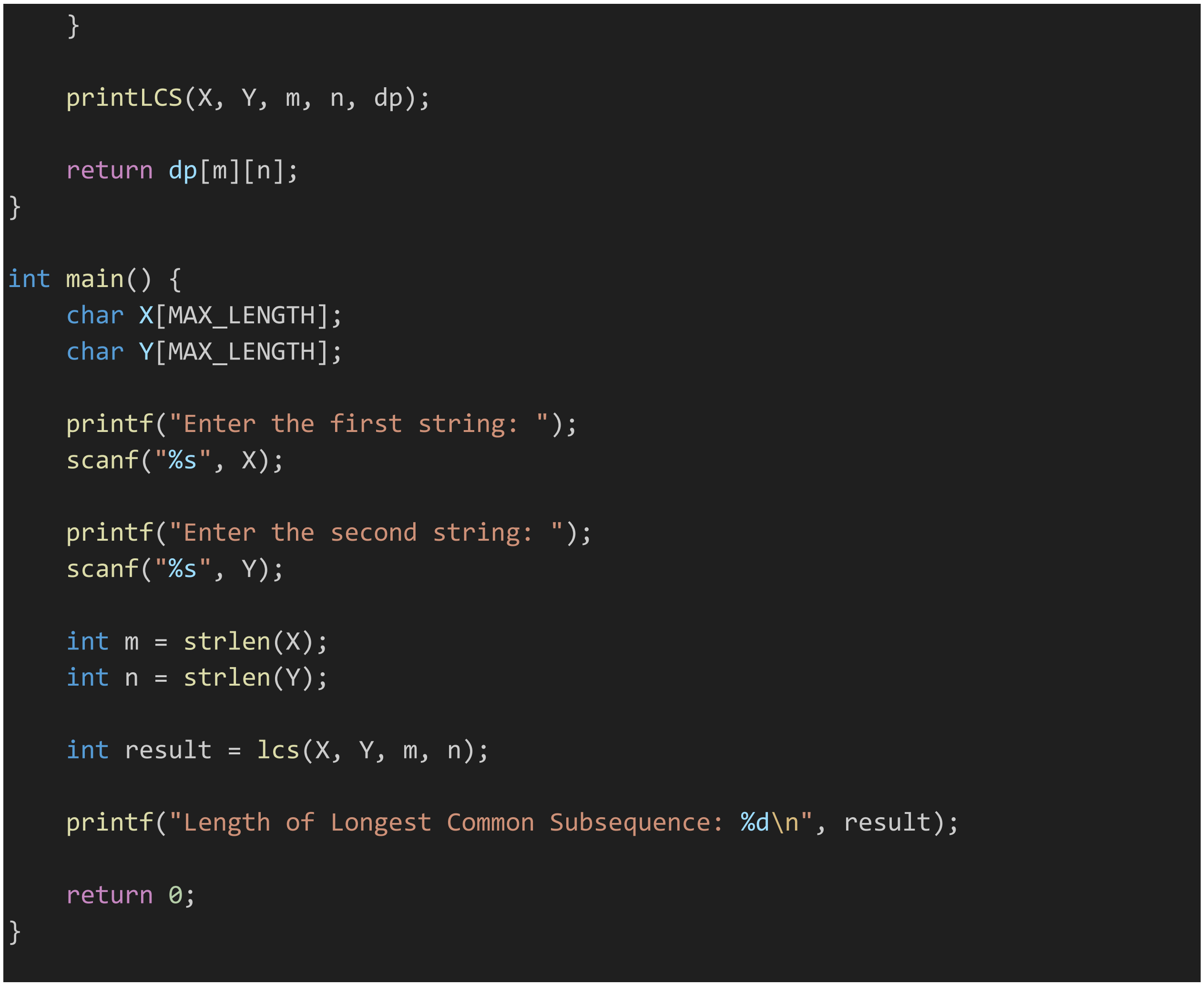
```
function printLCS( char*X, char*Y, m, n,
                        dp )
                                          22BCE3939
int i = m, j = n
index set to dp[m][n];
Let lcs be the string that stores
  substring
While i > 0 and j > 0
  do
    if x[i-1] is equal to Y[j-1]
        set lcs[index-1] to x[i-1]
          i = i - 1
          j = j - 1
          index = index - 1
    else if dp[i-1][j] > dp[i][j-1]
          i = i - 1
    else
          j = j - 1

print " Longest common subsequence" +
                 lcs
```

```
procedure main()
    X, Y                22BCE3939
    initialise result to 0
    print "Enter the first string"
    read X
    print "Enter the second string
    read Y
    result = LCS(X, Y)
    print "Length of LCS": , result
```

Source Code:

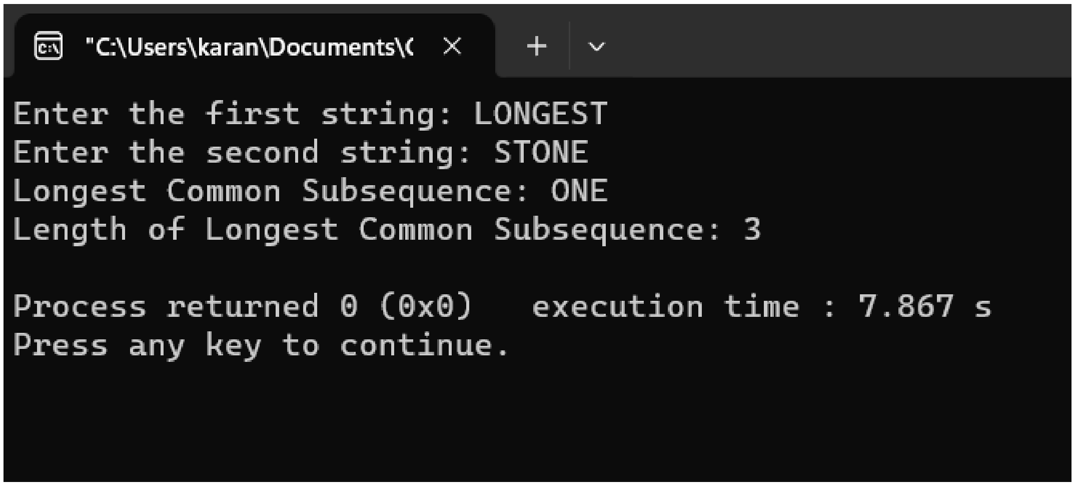


```c
#include <stdio.h>
#include <string.h>

#define MAX_LENGTH 100

int max(int a, int b) {
    return (a > b) ? a : b;
}

void printLCS(char* X, char* Y, int m, int n, int dp[MAX_LENGTH][MAX_LENGTH]) {
    int i = m, j = n;
    int index = dp[m][n];
    char lcs[MAX_LENGTH];

    while (i > 0 && j > 0) {
        if (X[i - 1] == Y[j - 1]) {
            lcs[index - 1] = X[i - 1];
            i--;
            j--;
            index--;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }

    printf("Longest Common Subsequence: %s\n", lcs);
}

int lcs(char* X, char* Y, int m, int n) {
    int dp[MAX_LENGTH][MAX_LENGTH];

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                dp[i][j] = 0;
            } else if (X[i - 1] == Y[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
```

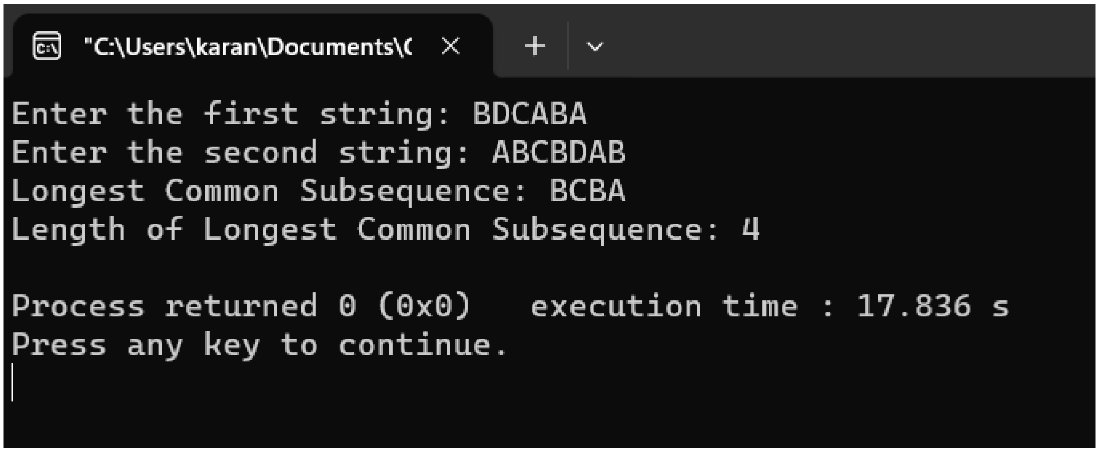


```c
    }

    printLCS(X, Y, m, n, dp);

    return dp[m][n];
}

int main() {
    char X[MAX_LENGTH];
    char Y[MAX_LENGTH];

    printf("Enter the first string: ");
    scanf("%s", X);

    printf("Enter the second string: ");
    scanf("%s", Y);

    int m = strlen(X);
    int n = strlen(Y);

    int result = lcs(X, Y, m, n);

    printf("Length of Longest Common Subsequence: %d\n", result);

    return 0;
}
```

Output:



```
Enter the first string: LONGEST
Enter the second string: STONE
Longest Common Subsequence: ONE
Length of Longest Common Subsequence: 3

Process returned 0 (0x0)   execution time : 7.867 s
Press any key to continue.
```



```
Enter the first string: BDCABA
Enter the second string: ABCBDAB
Longest Common Subsequence: BCBA
Length of Longest Common Subsequence: 4

Process returned 0 (0x0)   execution time : 17.836 s
Press any key to continue.
```

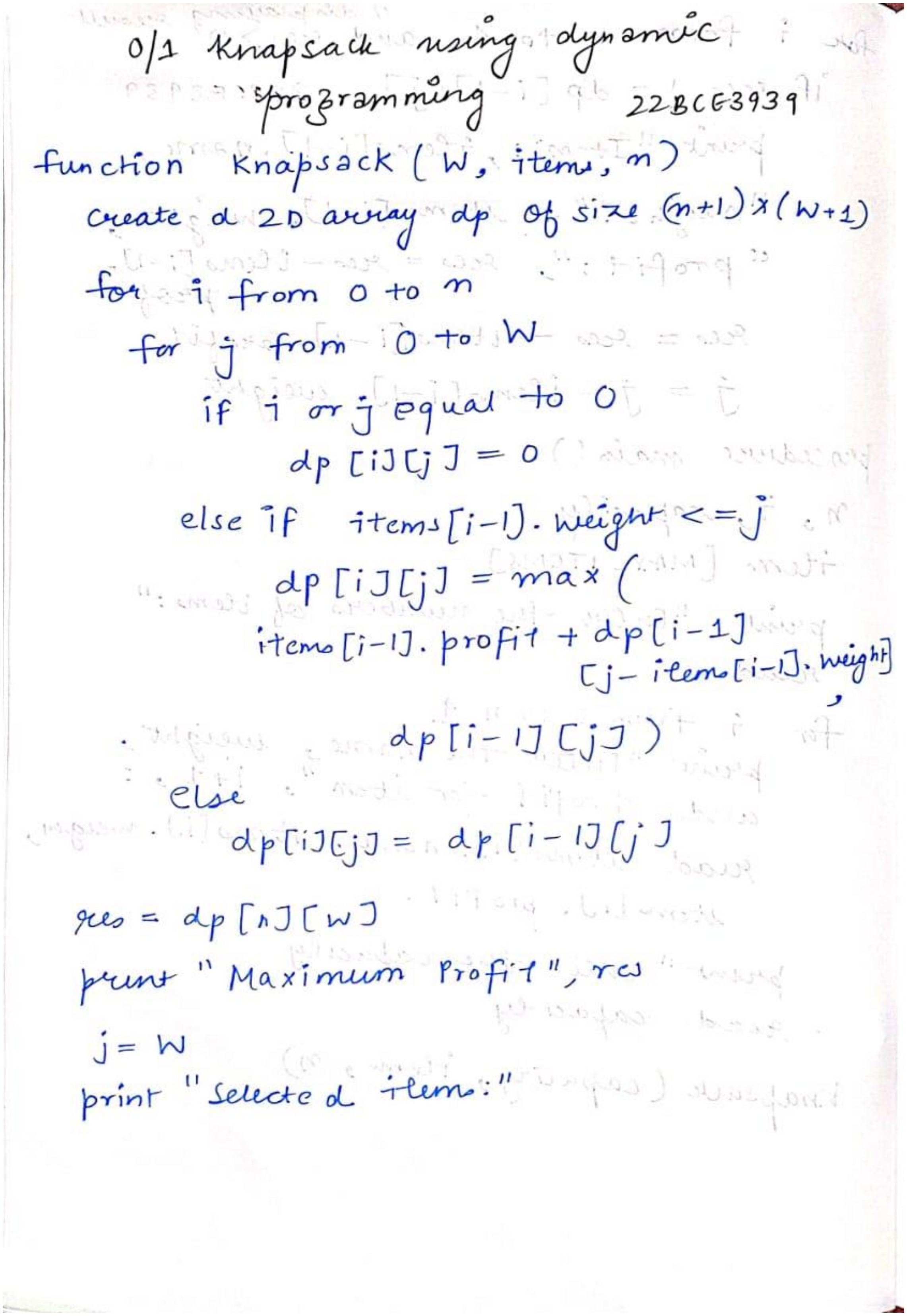
Question 4:

0/1 knapsack problem using DP approach.

Problem Statement:

The goal is to fill your knapsack with a maximum total value without exceeding the maximum weight capacity of the knapsack. The knapsack has a fixed capacity, and each item can either be selected (1) or rejected (0).

Pseudocode:



0/1 Knapsack using dynamic programming 22BCE3939

```
function Knapsack ( W, items, n)
  create a 2D array dp of size (n+1) x (W+1)
  for i from 0 to n
    for j from 0 to W
      if i or j equal to 0
          dp [i][j] = 0
      else if  items[i-1]. weight <= j
          dp [i][j] = max (
            items[i-1]. profit + dp[i-1]
                          [j- items[i-1]. weight],
                    dp [i-1] [j] )
      else
          dp[i][j] = dp [i- 1][j ]
res = dp [n][W]
print "Maximum Profit", res
j= W
print "Selected items:"
```

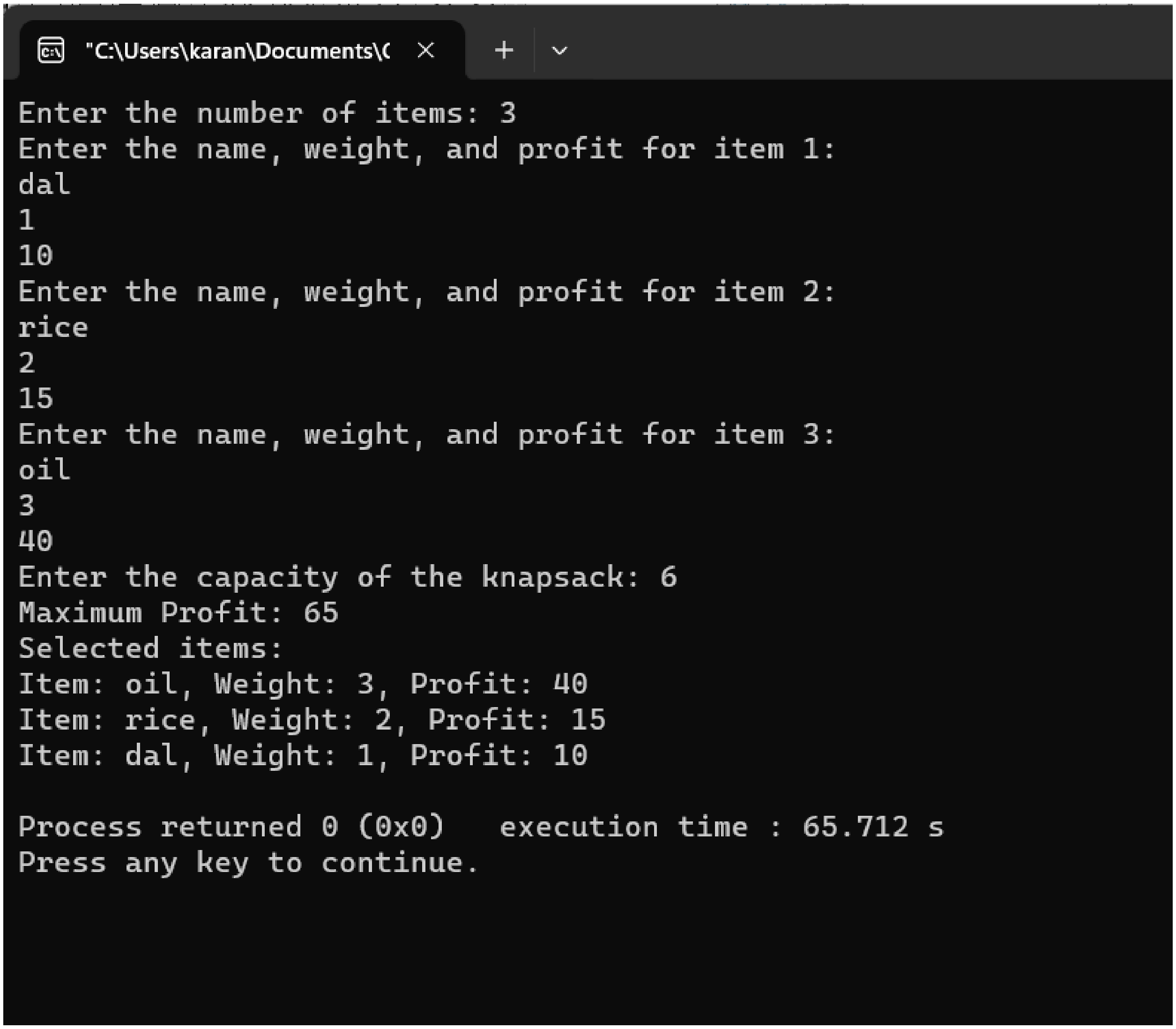
```
for i from n to 1 and res > 0   // displaying result
    if res != dp [i-1][j]        22BCE3939
        print "Item" : , items[i-1].name,
        "Weight :" , items [i-1]. weight,
        "profit :", res = res - items [i-1].
                                        profit
        res = res - items [i-1]. profit
        j = j - items [i-1]. weight
procedure main ()
    n, i, capacity
    items [MAX_ITEMS]
    print "Enter the numbers of item :"
    read n
    for i from 0 to n-1
        print "Enter the name, weight,
        and profit for item", i+1, :
        read items [i]. name, items [i]. weight,
            item [i]. profit.
    print " Enter the capacity
    read capacity
    Knapsack (capacity, items, n)
```

Source Code:

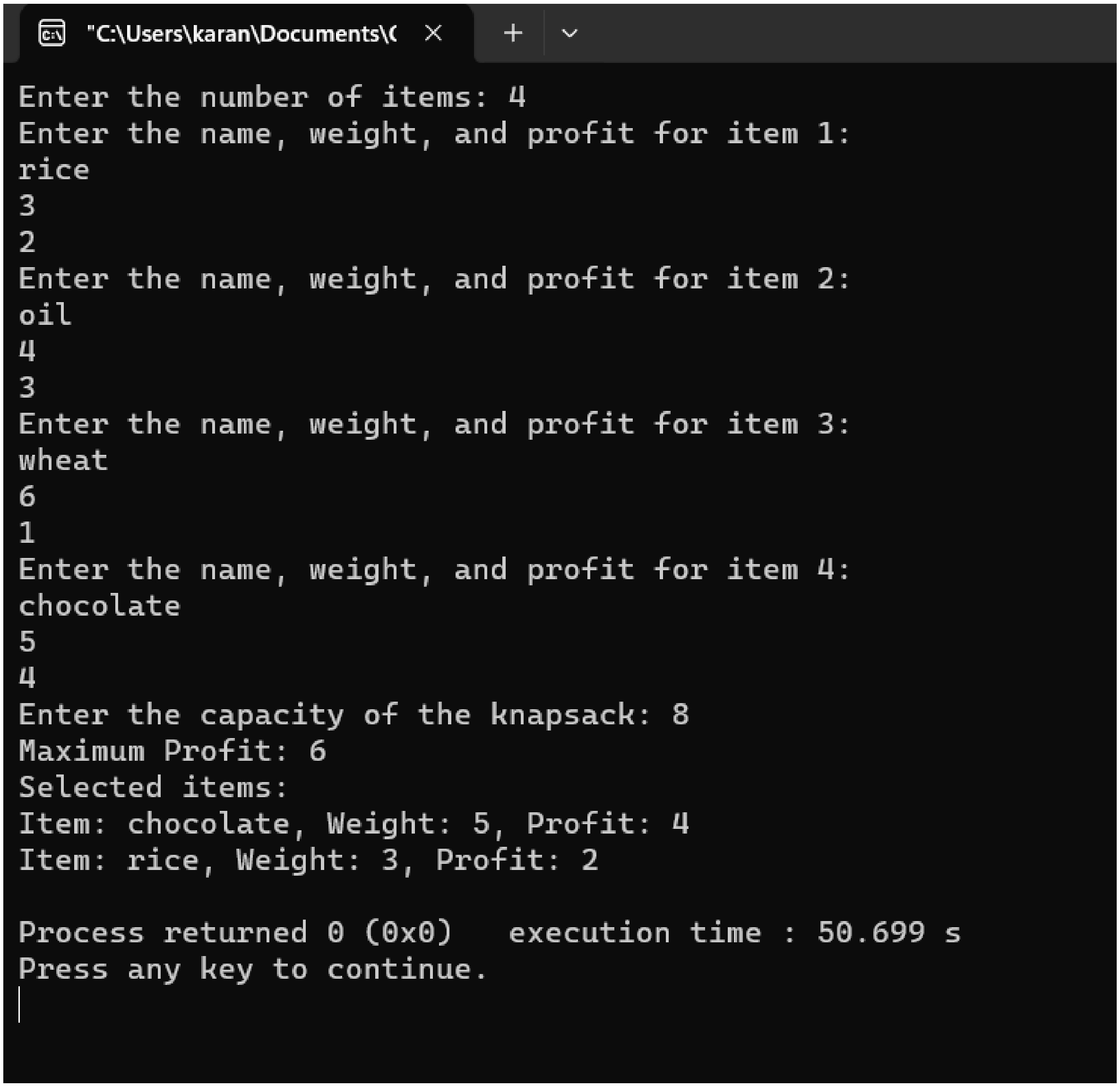
```
#include <stdio.h>
#include <stdlib.h>

#define MAX_ITEMS 100

// Structure to represent an item
struct Item {
    char name[20];
    int weight;
    int profit;
};

// Function to find the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to perform 0/1 knapsack using dynamic programming
void knapsack(int W, struct Item items[], int n) {
    int i, w;
    int dp[MAX_ITEMS + 1][W + 1];

    // Build table dp[][] in bottom-up manner
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (items[i - 1].weight <= w)
                dp[i][w] = max(items[i - 1].profit + dp[i - 1][w - items[i -
1].weight], dp[i - 1][w]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }

    // Print the selected items and maximum profit
    int res = dp[n][W];
    printf("Maximum Profit: %d\n", res);

    w = W;
    printf("Selected items:\n");

    for (i = n; i > 0 && res > 0; i--) {
```

```c
        if (res != dp[i - 1][w]) {
            printf("Item: %s, Weight: %d, Profit: %d\n", items[i - 1].name,
items[i - 1].weight, items[i - 1].profit);
            res = res - items[i - 1].profit;
            w = w - items[i - 1].weight;
        }
    }
}

int main() {
    int n, i, capacity;

    printf("Enter the number of items: ");
    scanf("%d", &n);

    struct Item items[MAX_ITEMS];

    for (i = 0; i < n; i++) {
        printf("Enter the name, weight, and profit for item %d:\n", i + 1);
        scanf("%s %d %d", items[i].name, &items[i].weight, &items[i].profit);
    }

    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &capacity);

    knapsack(capacity, items, n);

    return 0;
}
```

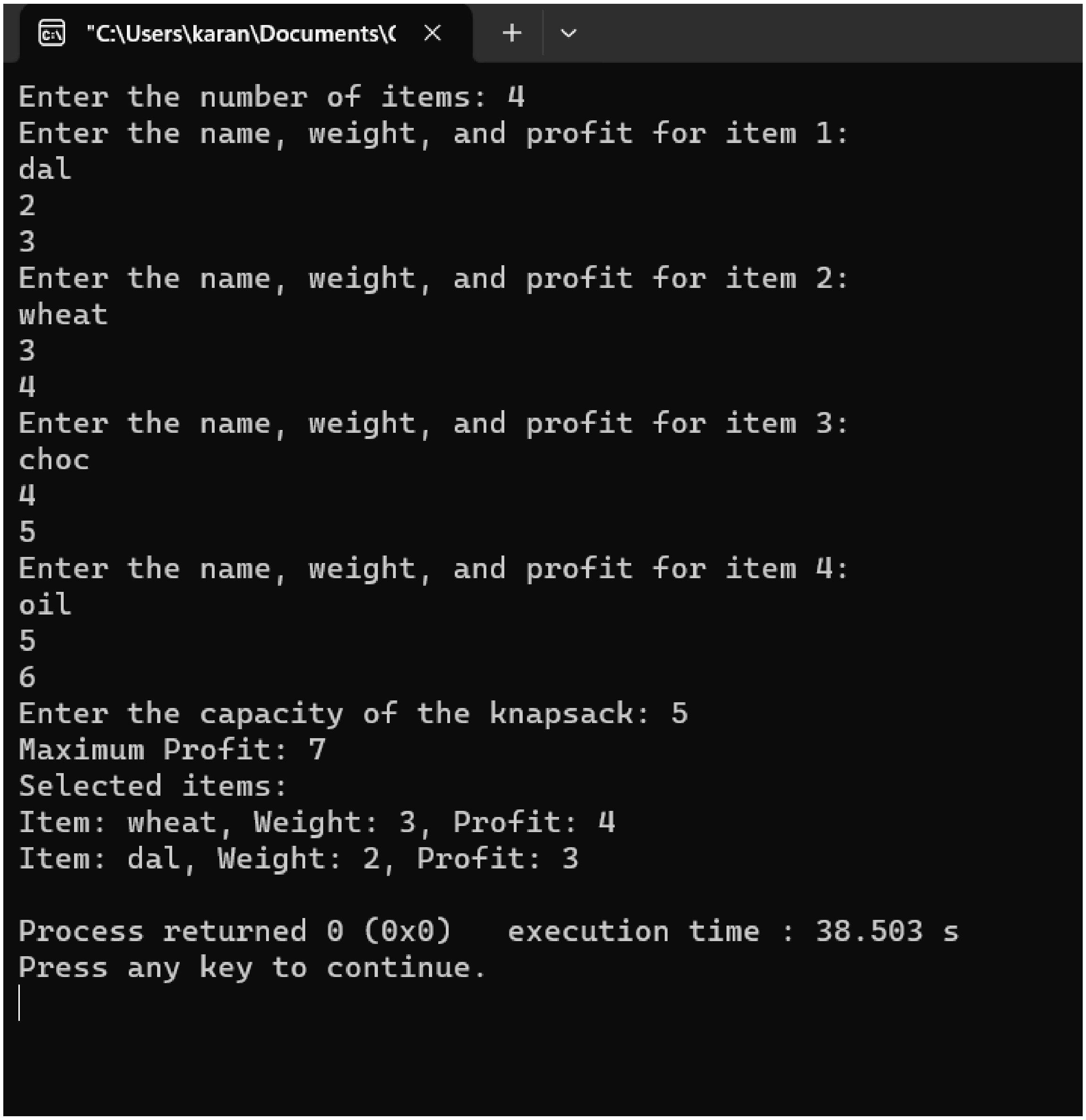
Output:



```
Enter the number of items: 3
Enter the name, weight, and profit for item 1:
dal
1
10
Enter the name, weight, and profit for item 2:
rice
2
15
Enter the name, weight, and profit for item 3:
oil
3
40
Enter the capacity of the knapsack: 6
Maximum Profit: 65
Selected items:
Item: oil, Weight: 3, Profit: 40
Item: rice, Weight: 2, Profit: 15
Item: dal, Weight: 1, Profit: 10

Process returned 0 (0x0)   execution time : 65.712 s
Press any key to continue.
```



```
Enter the number of items: 4
Enter the name, weight, and profit for item 1:
rice
3
2
Enter the name, weight, and profit for item 2:
oil
4
3
Enter the name, weight, and profit for item 3:
wheat
6
1
Enter the name, weight, and profit for item 4:
chocolate
5
4
Enter the capacity of the knapsack: 8
Maximum Profit: 6
Selected items:
Item: chocolate, Weight: 5, Profit: 4
Item: rice, Weight: 3, Profit: 2

Process returned 0 (0x0)   execution time : 50.699 s
Press any key to continue.
```



```
Enter the number of items: 4
Enter the name, weight, and profit for item 1:
dal
2
3
Enter the name, weight, and profit for item 2:
wheat
3
4
Enter the name, weight, and profit for item 3:
choc
4
5
Enter the name, weight, and profit for item 4:
oil
5
6
Enter the capacity of the knapsack: 5
Maximum Profit: 7
Selected items:
Item: wheat, Weight: 3, Profit: 4
Item: dal, Weight: 2, Profit: 3

Process returned 0 (0x0)   execution time : 38.503 s
Press any key to continue.
```

Question 5:

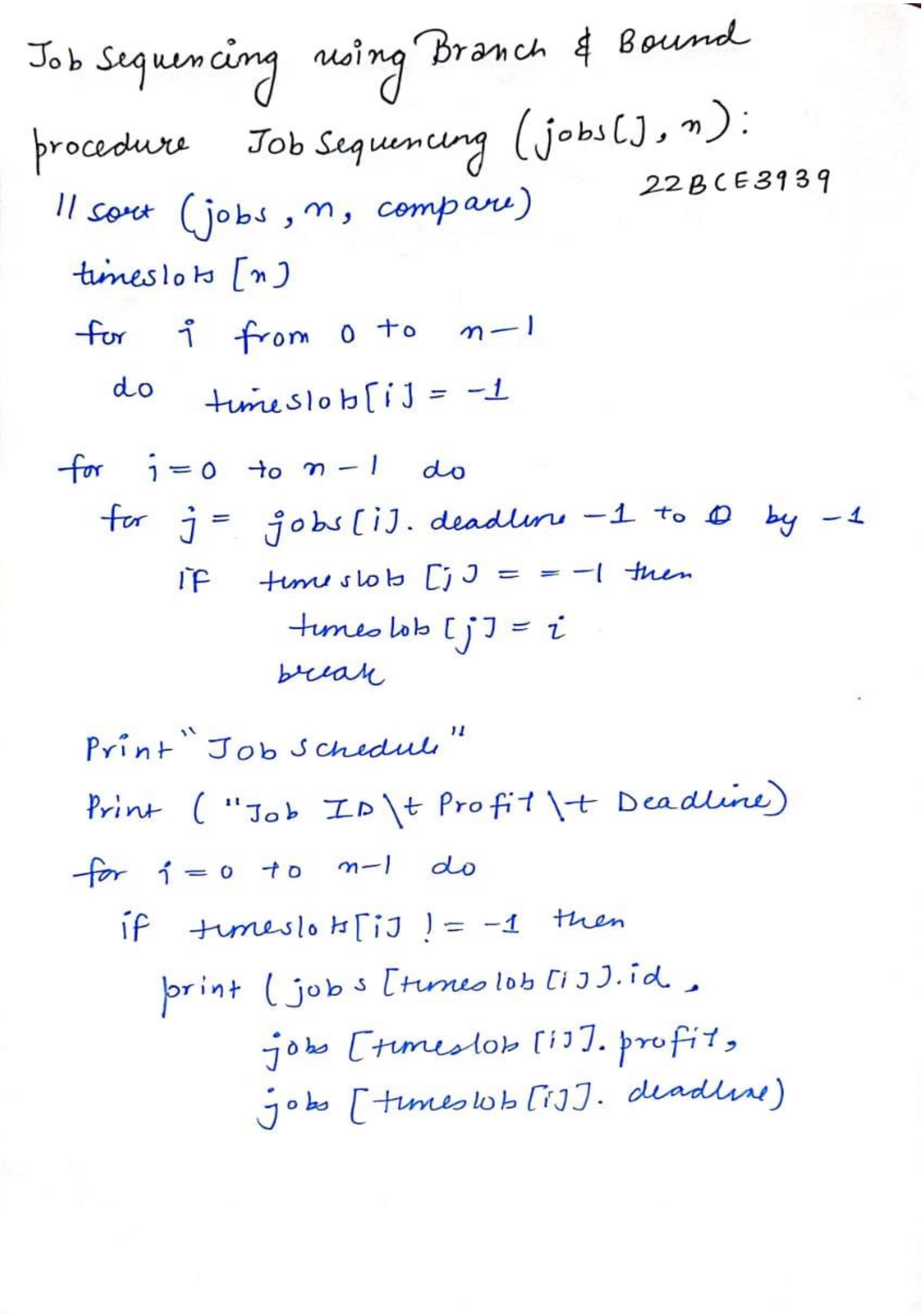
Job selection using branch and bound.

Problem Statement:

Consider a set of n jobs, each characterized by a unique job ID, a profit value, and a deadline by which it needs to be completed. The goal is to schedule the jobs in a way that maximizes the total profit.

Each job takes one unit of time to complete, and only one job can be scheduled at a time. If a job is not completed by its deadline, it is considered missed, and no profit is earned for that job.

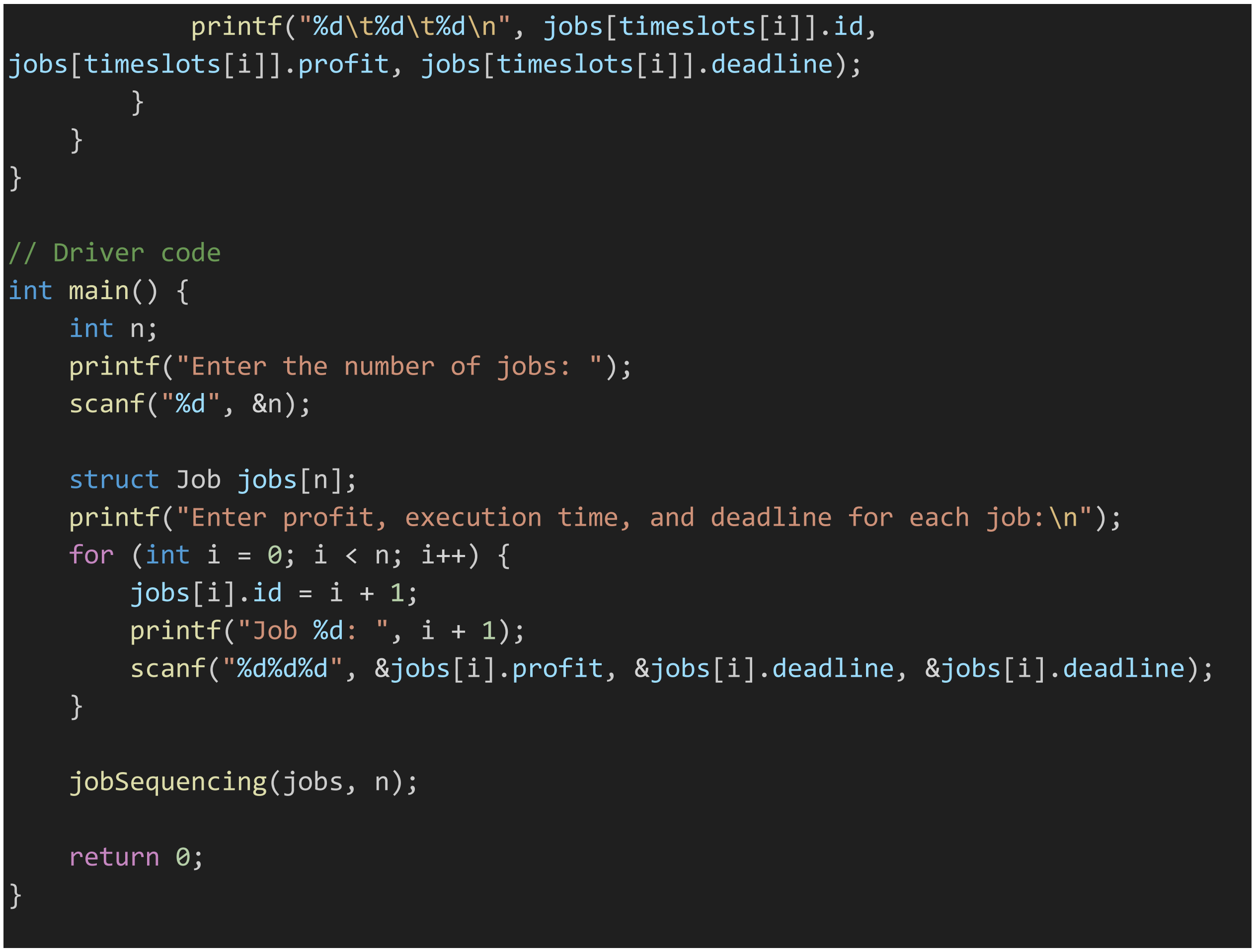
Pseudo-code:



```
Job Sequencing using Branch & Bound

procedure Job Sequencing (jobs[], n):
                                              22BCE3939
 // sort (jobs, n, compare)
  timeslots [n]
  for i from 0 to n-1
    do  timeslot[i] = -1

 for i=0 to n-1 do
   for j = jobs[i]. deadline -1 to 0 by -1
        if  timeslot [j] = = -1 then
              timeslot [j] = i
            break

  Print "Job Schedule"
  Print ( "Job ID\t Profit \t Deadline)
  for i = 0 to n-1 do
    if timeslots[i] != -1 then
       print ( jobs [timeslot [i]].id,
               jobs [timeslot [i]]. profit,
               jobs [timeslot [i]]. deadline)
```

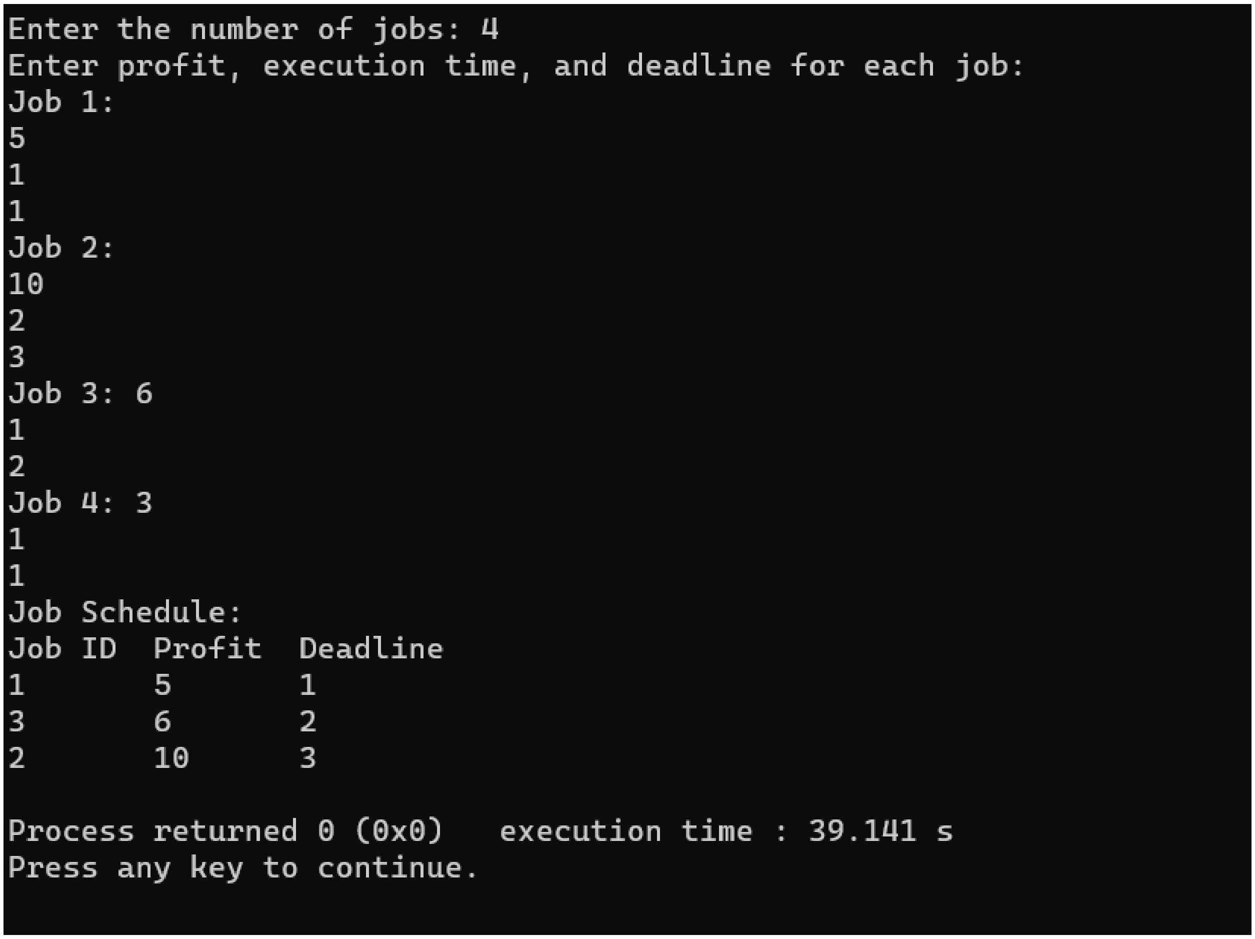
Source code:

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

// Structure to represent a job
struct Job {
    int id;
    int profit;
    int deadline;
};

// Function to compare jobs based on profit
int compare(const void* a, const void* b) {
    return ((struct Job*)b)->profit - ((struct Job*)a)->profit;
}

// Function to find the maximum profit job schedule using Greedy method
void jobSequencing(struct Job jobs[], int n) {
    // Sort jobs based on profit in non-increasing order
    qsort(jobs, n, sizeof(struct Job), compare);

    // Array to keep track of time slots
    int timeslots[n];

    // Initialize all slots to be empty
    for (int i = 0; i < n; i++)
        timeslots[i] = -1;

    // Iterate through each job and assign it to the latest possible time slot
before its deadline
    for (int i = 0; i < n; i++) {
        for (int j = (jobs[i].deadline - 1); j >= 0; j--) {
            if (timeslots[j] == -1) {
                timeslots[j] = i;
                break;
            }
        }
    }

    // Display the schedule
    printf("Job Schedule:\n");
    printf("Job ID\tProfit\tDeadline\n");
    for (int i = 0; i < n; i++) {
        if (timeslots[i] != -1) {
```



```
            printf("%d\t%d\t%d\n", jobs[timeslots[i]].id,
jobs[timeslots[i]].profit, jobs[timeslots[i]].deadline);
        }
    }
}

// Driver code
int main() {
    int n;
    printf("Enter the number of jobs: ");
    scanf("%d", &n);

    struct Job jobs[n];
    printf("Enter profit, execution time, and deadline for each job:\n");
    for (int i = 0; i < n; i++) {
        jobs[i].id = i + 1;
        printf("Job %d: ", i + 1);
        scanf("%d%d%d", &jobs[i].profit, &jobs[i].deadline, &jobs[i].deadline);
    }

    jobSequencing(jobs, n);

    return 0;
}
```

Output:



```
Enter the number of jobs: 4
Enter profit, execution time, and deadline for each job:
Job 1:
5
1
1
Job 2:
10
2
3
Job 3: 6
1
2
Job 4: 3
1
1
Job Schedule:
Job ID  Profit  Deadline
1       5       1
3       6       2
2       10      3

Process returned 0 (0x0)   execution time : 39.141 s
Press any key to continue.
```