Lab -

DIP : Design and Analysis of Algorithm

AIM : Implement the following Dynamic Programming Problems

(1) 0/1 Knapsack Problem

(2) Assembly line Scheduling Problem

Name : Ambalia Harshit

Department : Computer Engineering (M.Tech sem II)

Roll no : MT001

Date : Jan 2024

Program 01 : 0/1 Knapsack Problem

- Description :

The 0/1 knapsack problem using dynamic approach is a method for solving the 0/1 knapsack problem, which is a combinatorial optimization problem where the goal is to find the combination of items that maximizes the total value of the knapsack.

The dynamic approach to solving the problem involves using dynamic programming to construct a table that stores the maximum value that can be achieved for a given knapsack capacity and set of items. The table is filled using a bottom-up approach, starting with the smallest subproblems and building up to the overall problem.

The table is constructed by iterating through all the items and for each item, considering two cases: either including the item or not including it. The maximum value for a given knapsack capacity and set of items is the maximum of the two cases.

The table can then be used to determine the optimal combination of items to include in the knapsack. The optimal solution can be reconstructed by starting from the last cell of the table and tracing back the decisions that were made at each step to determine which items were included in the solution.

The time complexity of this approach is O(nW) where n is the number of items and W is the knapsack's capacity.

- Algorithm :

```
Dynamic-0-1-knapsack (v, w, n, W)
        for w = 0 to W do
                c[0, w] = 0
        for i = 1 to n do
                c[i, 0] = 0
                for w = 1 to W do
                        if wi ≤ w then
                                if vi + c[i-1, w-wi] then
                                        c[i, w] = vi + c[i-1, w-wi]
                                else c[i, w] = c[i-1, w]
                        Else
                                c[i, w] = c[i-1, w]
```

- Code :
  - Following code does not represent the given algorithm, instead it recursively finds the solution while utilizing the memory table.

```c
// 0/1 knapsack problem

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

int** initialize_MemoryTable(int n, int kw) {
   int** t = (int **)malloc((n + 1) * sizeof(int *));
   for (int i = 0; i <= n; i++) {
       t[i] = (int *)malloc((kw + 1) * sizeof(int));
       for (int j = 0; j <= kw; j++) {
           t[i][j] = -1;
       }
   }
   return t;
}

int dp_knapsack(int length, int weight[], int value[], int capacity, int
**t) {
   if(capacity==0 || length==0) {
       return 0;
   }

   // If already computed, no need to compute again
   if(t[length][capacity]!=-1) {
       return t[length][capacity];
   }

   else if(weight[length-1]<=capacity) {
       int temp1 = value[length-1]+dp_knapsack(length-1, weight, value,
capacity-weight[length-1], t);
       int temp2 = dp_knapsack(length-1, weight, value, capacity, t);
       if(temp1<temp2) {
           t[length][capacity] = temp2;
           return temp2;
       }
```

```c
        else{
            t[length][capacity] = temp1;
            return temp1;
        }
    }
    else if(weight[length-1]>capacity) {
        int temp2 = dp_knapsack(length-1, weight, value, capacity, t);
        t[length][capacity] = temp2;
        return temp2;
    }
}

int main() {
    int n = 4;
    int kw = 8;
    int weight[] = {2, 3, 4, 5};
    int profit[] = {10, 20, 50, 60};
    int **t = initialize_MemoryTable(n, kw);

    int temp = dp_knapsack(n, weight, profit, kw, t);
    printf("Final : %d\n", temp);

    for (int i=0;i<n;i++){
        for (int j=0;j<kw;j++){
            printf("%d ", t[i][j]);
        }
        printf("\n");
    }
}
```

- Output Screen-shots :

```
hr@Edith:~/Documents/Semester_10/Lab_DAA/Lab_03$ cd "/h
ex01.c -o ex01 && "/home/hr/Documents/Semester_10/Lab_D
Final : 80
-1 -1 -1 -1 -1 -1 -1 -1
-1 0 -1 10 10 10 -1 -1
-1 -1 -1 20 20 -1 -1 -1
-1 -1 -1 20 -1 -1 -1 -1
hr@Edith:~/Documents/Semester_10/Lab_DAA/Lab_04$
```

Program 02 : Assembly line Scheduling Problem

- Description :

The Assembly Line Scheduling Problem (ALSP) using dynamic approach is a method for solving the ALSP, which is a combinatorial optimization problem where the goal is to minimize the total completion time of a set of jobs that must be processed on a number of assembly lines while satisfying all the constraints.

The dynamic approach to solving the problem involves using dynamic programming to construct a table that stores the minimum completion time and the corresponding task sequence for a given set of jobs and assembly lines.

The table is filled using a bottom-up approach, starting with the smallest subproblems and building up to the overall problem. The table is constructed by iterating through all the jobs and for each job, considering two cases: either starting the job from the first assembly line or from the second assembly line.

The minimum completion time for a given job is the minimum of the two cases. The table can then be used to determine the optimal task sequence for each job that satisfies all the constraints and minimize the total completion time.

The optimal solution can be reconstructed by starting from the last cell of the table and tracing back the decisions that were made at each step to determine the task sequence for each job.

The time complexity of this approach is O(n^2*m) where n is the number of jobs and m is the number of assembly lines. It's worth noting that ALSP is an NP-hard problem, so this problem is hard to solve optimally and there are different other methods to approach this problem such as Genetic Algorithm, Simulated Annealing and other heuristic methods that have been proposed to solve this problem.

- Algorithm :

Algorithm ASSEMBLY_LINE_SCHEDULING(n, e, a, t, x)

```
// n is number of stations on both assembly
// e is array of entry time on assembly
// a is array of assembly time on given station
// t is array of the time required to change assembly line
//x is array of exit time from assembly
```

```
    f1[1] ← e1 + a1,1
    f2[1] ← e2 + a2,1
    for j ← 2 to n do
        if f1[j – 1] + a1, j ≤ f2[j – 1] + t2, j – 1 + a2, j then
            f1[j] ← f1[j – 1] + a1, j
            L1[j] ← 1
        Else
            f1[j] ← f2[j – 1] + t2,j – 1 + a1, j
            L1[j] ← 2
        End
        if f2[j – 1] + a2, j ≤ f1[j – 1] + t1, j – 1 + a2, j then
            f2[j] ← f2[j – 1] + a2, j
            l2[j] ← 2
        else
            f2[j] ← f1[j – 1] + t1, j – 1 + a2, j
            l2[j] ← 1
        End
        if f1[n] + x1 ≤ f2[n] + x2 then
            F* ← f1[n] + x1
            L* ← 1
        else
            F* ← f2[n] + x2
            L* ← 2
        end
    End

Algorithm PRINT_STATIONS(l, n)
    i ← 1*
    print "line " i ", station " n
    for j ← n downto 2 do
        i ← li[ j ]
        print "line " i ",station " j – 1
    end
```

- Code :
  - Following code does not represent the given algorithm, instead it recursively finds the solution while utilizing the memory table.

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

#define NUM_STATIONS 4

// Initialize the memory table
int** initialize_MemoryTable(int n, int kw) {
   int** t = (int **)malloc((n + 1) * sizeof(int *));
   for (int i = 0; i <= n; i++) {
       t[i] = (int *)malloc((kw + 1) * sizeof(int));
       for (int j = 0; j <= kw; j++) {
           t[i][j] = -1;
       }
   }
   return t;
}

// Calculate the minimum time to complete the assembly
int minTimeMemo(int process_time[][NUM_STATIONS], int
transfer_time[][NUM_STATIONS], int entry_time[], int exit_time[], int
stations, int line, int station, int **memo) {
   if ( station==0 ) {
       return entry_time[line] + process_time[line][0];
   }

   // Check if the result is already memoized
   if ( memo[line][station]!=-1 ) {
       return memo[line][station];
   }

   int timeFromPrevLine = minTimeMemo(process_time, transfer_time,
entry_time, exit_time, stations, 1-line, station-1, memo) +
transfer_time[1 - line][station - 1];
```

```
    int timeFromSameLine = minTimeMemo(process_time, transfer_time,
entry_time, exit_time, stations, line, station-1, memo) +
process_time[line][station];
    if(timeFromPrevLine < timeFromSameLine) {
        memo[line][station] = timeFromPrevLine;
    }
    else{
        memo[line][station] = timeFromSameLine;
    }

    return memo[line][station];
}

int assembly_scheduling(int process_time[][NUM_STATIONS], int
transfer_time[][NUM_STATIONS], int entry_time[], int exit_time[], int
stations) {

    int **memo = initialize_MemoryTable(2, NUM_STATIONS);

    int timeLine1 = minTimeMemo(process_time, transfer_time, entry_time,
exit_time, stations, 0, stations - 1, memo) + exit_time[0];
    int timeLine2 = minTimeMemo(process_time, transfer_time, entry_time,
exit_time, stations, 1, stations - 1, memo) + exit_time[1];
    if(timeLine1 < timeLine2) {
        return timeLine1;
    }
    else{
        return timeLine2;
    }
}

int main() {
    int process_time[2][NUM_STATIONS] = {
        {4, 5, 3, 2},
        {2, 10, 1, 4}
    }; // Time at each station on each line

    int transfer_time[2][NUM_STATIONS] = {
        {0, 7, 4, 5},
        {0, 9, 2, 8}
```

```c
    }; // Time to transfer between stations

    int entry_time[2] = {10, 12};
    int exit_time[2] = {18, 7};
    int stations = NUM_STATIONS;

    int answer = assembly_scheduling(process_time, transfer_time,
entry_time, exit_time, stations);

    printf("Minimum time to complete : %d\n", answer);

    return 0;
}
```

- Output Screen-Shots :

```
hr@Edith:~/Documents/Semester_10/Lab_DAA/Lab_04$ cd "/h
ex02.c -o ex02 && "/home/hr/Documents/Semester_10/Lab_D
Minimum time to complete : 26
hr@Edith:~/Documents/Semester_10/Lab_DAA/Lab_04$
```

- Output Screen-shots :

```
hr@Edith:~/Documents/Semester_10/Lab_DAA/Lab_03$ cd "/home/hr/Docum
ex02.c -o ex02 && "/home/hr/Documents/Semester_10/Lab_DAA/Lab_03/"e
Value : 141
Coins Are : 2 5 10 25 50
We need 2 Coin of rupees 50 = 100
We need 1 Coin of rupees 25 = 25
We need 1 Coin of rupees 10 = 10
We need 1 Coin of rupees 5 = 5
Can generate change for 140 only with given coins, 1 is remaining.
hr@Edith:~/Documents/Semester_10/Lab_DAA/Lab_03$
```

```
hr@Edith:~/Documents/Semester_10/Lab_DAA/Lab_03$ cd "/home/hr/Docum
ex02.c -o ex02 && "/home/hr/Documents/Semester_10/Lab_DAA/Lab_03/"e
Value : 142
Coins Are : 2 5 10 25 50
We need 2 Coin of rupees 50 = 100
We need 1 Coin of rupees 25 = 25
We need 1 Coin of rupees 10 = 10
We need 1 Coin of rupees 5 = 5
We need 1 Coin of rupees 2 = 2
hr@Edith:~/Documents/Semester_10/Lab_DAA/Lab_03$
```