Dynamic Programming Assignment

Shashwat Gupta (14IE10028)

March 31, 2016

1 Satisfying parenthesisation of Boolean expressions /w OR (Problem 7)

The following is the code for my function which returns the number of parenthesis possible. I make use of two 2-D arrays and store the bottom-up table in them for true and false respectively. The tables are then evaluated using a formula for OR gate and the expression is checked for true or false and then updated accordingly.

The code

```
int countParenth(char expression[], int n)
{
    int i,j,g,gap;
    int F[n][n], T[n][n];
    for (i = 0; i < n; i++)
    {
        F[i][i] = (expression[i] == 'F')? 1: 0;
        T[i][i] = (expression[i] == 'T')? 1: 0;
    }
    for (gap=1; gap<n; ++gap)
    {
        for (i=0, j=gap; j<n; ++i, ++j)
        {
            T[i][j] = 0;
            F[i][j] = 0;
            for (g=0; g<gap; g++)
            {
                // Find place of parenthesization using current
    value of gap
                int k = i + g;
                // Store Total[i][k] and Total[k+1][j]
                int tik = T[i][k] + F[i][k];
                int tkj = T[k+1][j] + F[k+1][j];
                // Follow the recursive formula
                F[i][j] += F[i][k]*F[k+1][j];
                T[i][j] += (tik*tkj - F[i][k]*F[k+1][j]);
            }
        }
    }
    return T[0][n-1];
```

```
}
```

Listing 1: Parenthesis

Complexity Analysis
These are the costs:

1. Time Complexity $\in O(n^3)$.
2. Space Complexity $\in O(n^2)$.

2 Knapsack packing with item repetition (Problem 11)

We accept the details for all the items and the knapsack in hand. We then make use of an array K which store the maximum values for each capacity i in K[i]. thus, our objective is to obtain K[capacity]; For 0 capacity, max value = 0, thus K[0]=0 for all other i, first K[i] is initialised to K[i-1] as this is obvious and then we check with every element in hand along with the K of (total capacity - capacity of that element in hand). The maximum is chosen and K[i] is updated. This is repeated for all n.

The following is the code

```
int main()
{
    int i,j,capacity,n;
    printf("Enter capacity of knapsack\n");
    scanf("%d",&capacity);
    printf("Enter number of items\n");
    scanf("%d",&n);
    int val[n];
    int size[n];
    for (i = 0; i < n; ++i)
    {
        printf("Value for item %d = ",i+1);
        scanf("%d",&val[i]);
        printf("Weight for item %d = ",i+1);
        scanf("%d",&size[i]);
    }
    int K[capacity+1];
    K[0] = 0;
    for (i = 1; i <= capacity; ++i)
    {
        K[i]=K[i-1];
        for (j = 0; j < n; ++j)
        {
            if((i-size[j])>=0 && K[i]<(K[i-size[j]]+val[j]))
                K[i]=K[i-size[j]]+val[j];
        }
    }
    printf("Maximum Value = %d\n", K[capacity]);
    return 0;
```

```
30 }
```

Listing 2: knapsack

Complexity Analysis
These are the costs:

1. Time Complexity $\in O(n * capacity)$.
2. Space Complexity $\in O(capacity)$.