# 1 Dynamic Programming

## 1.1 Knapsack

```cpp

vector<vector<ll>> DP;
vector<ll> Weights;
vector<ll> Values;

ll Knapsack(int w, int i)
{
    if (w == 0 or i == -1)
        return 0;
    if (DP[w][i] != -1)
        return DP[w][i];
    if (Weights[i] > w)
        return DP[w][i] = Knapsack(w, i - 1);
    return DP[w][i] = max(Values[i] + Knapsack(w - Weights[i], i - 1),
         Knapsack(w, i - 1));
}
```

## 1.2 Matrix Chain Multiplication

```cpp

vector<vector<ii>> DP; //Pair value, op result
int n;                 //Size of DP (i.e. i,j<n)
ii op(ii a, ii b)
{
    return {a.first + b.first + a.second * b.second, (a.second + b.
         second) % 100}; //Second part MUST be associative, first part
         is cost function
}

ii MCM(int i, int j)
{
    if (DP[i][j].first != -1)
        return DP[i][j];
    int ans = 1e9; //INF
    int res;
    repx(k, i + 1, j)
    {
        ii temp = op(MCM(i, k), MCM(k, j));
        ans = min(ans, temp.first);
        res = temp.second;
    }
    return DP[i][j] = {ans, res};
}

void fill()
{
    DP.assign(n, vector<ii>(n, {-1, 0}));
    rep(i, n - 1) { DP[i][i + 1].first = 1; } // Pair op identity, cost
         (cost must be from input)
}
```

## 1.3 Longest Increasing Subsequence

```cpp

vi L;
vi vals;

int maxl = 1;

//Bottom up approach O(nlogn)
int lis(int n)
{
    L.assign(n, -1);
    L[0] = vals[0];
    repx(i, 1, n)
    {
        auto it = lower_bound(L.begin(), L.begin() + maxl, vals[i]);
        if (it == L.begin() + maxl)
        {
            L[maxl] = vals[i];
            maxl++;
        }
        else
            *it = vals[i];
    }
    return maxl;
}
```