# Dynamic programming

## Knapsack Problem

Q1.(change the programing language to c)

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

int compare(const void *a, const void *b) {
    double diff = ((double (*)[2])a)[0][0] - ((double (*)[2])b)[0][0];
    return (diff > 0) - (diff < 0);
}

int main() {
    int n;
    scanf("%d", &n);
    double wmax;
    scanf("%lf", &wmax);

    double a[n][2];
    for(int i = 0; i < n; i++) {
        double w, v;
        scanf("%lf %lf", &w, &v);
        double value_per_weight = w / v;

        a[i][0] = value_per_weight;
        a[i][1] = w;
    }

    qsort(a, n, sizeof(a[0]), compare);

    double ans = 0;

    for(int i = n - 1; i >= 0; i--) {
        double weight_to_take = (wmax < a[i][1]) ? wmax : a[i][1];
        ans += weight_to_take * a[i][0];
        wmax -= weight_to_take;
        if (wmax <= 0) break;
    }

    printf("%.16f\n", ans);

    return 0;
}
```

Q3.

2
3
Q6.
0 and 0
Q7.

```
#include <iostream>
#include <vector>
using namespace std;

int knapsack(int Wmax, vector<int>& weights, vector<int>& values, int N) {
    vector<vector<int>> dp(N + 1, vector<int>(Wmax + 1, 0));

    for (int i = 1; i <= N; ++i) {
        int Wi = weights[i - 1];
        int Vi = values[i - 1];
        for (int w = 1; w <= Wmax; ++w) {
            if (Wi <= w) {
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-Wi] + Vi);
            } else {
                dp[i][w] = dp[i-1][w];
            }
        }
    }

    return dp[N][Wmax];
}

int main() {
    int N, Wmax;
    cin >> N >> Wmax;

    vector<int> weights(N);
    vector<int> values(N);

    for (int i = 0; i < N; ++i) {
        cin >> weights[i] >> values[i];
    }

    int maxValue = knapsack(Wmax, weights, values, N);
    cout << maxValue << endl;

    return 0;
}
```

Q9.

```
#include <iostream>
#include <vector>
using namespace std;
```

```
bool subsetSumExists(int N, int X, vector<int>& A) {
    // dp[i][j] will be true if there is a subset of elements from A[0] to A[i-1] that has sum equal
to j
    vector<vector<bool>> dp(N + 1, vector<bool>(X + 1, false));

    // Base case: subset with sum 0 is always possible by taking no elements
    dp[0][0] = true;

    for (int i = 1; i <= N; ++i) {
        int currentValue = A[i - 1];
        for (int j = 0; j <= X; ++j) {
            dp[i][j] = dp[i - 1][j]; // exclude current element
            if (j >= currentValue) {
                dp[i][j] = dp[i][j] || dp[i - 1][j - currentValue]; // include current element
            }
        }
    }

    return dp[N][X];
}

int main() {
    int N, X;
    cin >> N >> X;

    vector<int> A(N);
    for (int i = 0; i < N; ++i) {
        cin >> A[i];
    }

    if (subsetSumExists(N, X, A)) {
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }

    return 0;
}
```

## Longest common subsequence

Q4.
Zero and Zero
Q5.

```
#include <iostream>
#include <vector>
#include <string>
```

```
using namespace std;

int longestCommonSubsequence(string S, string T) {
    int n = S.length();
    int m = T.length();

    // Create dp table with dimensions (n+1) x (m+1)
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

    // Fill dp table
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            if (S[i - 1] == T[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    // Length of LCS is stored in dp[n][m]
    return dp[n][m];
}

int main() {
    string S, T;
    cin >> S >> T;

    int result = longestCommonSubsequence(S, T);
    cout << result << endl;

    return 0;
}
```

Q6.
NM and NM

Q7.

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int longestPalindromicSubsequence(string S) {
    int n = S.length();

    // Create a dp table of size n x n
    vector<vector<int>> dp(n, vector<int>(n, 0));
```

```
    // Base case: single character is always a palindrome of length 1
    for (int i = 0; i < n; ++i) {
        dp[i][i] = 1;
    }

    // Build dp table
    for (int len = 2; len <= n; ++len) {
        for (int i = 0; i <= n - len; ++i) {
            int j = i + len - 1;
            if (S[i] == S[j]) {
                dp[i][j] = dp[i+1][j-1] + 2;
            } else {
                dp[i][j] = max(dp[i+1][j], dp[i][j-1]);
            }
        }
    }

    // Length of longest palindromic subsequence is dp[0][n-1]
    return dp[0][n-1];
}

int main() {
    string S;
    cin >> S;

    int result = longestPalindromicSubsequence(S);
    cout << result << endl;

    return 0;
}
```

## Warshall's and Floyd's algorithm

Q4.
0
Q5.

## Optimal Binary Search Trees

Q2.
11 and 1

Q6.
F[i] and 0

Q7.

```
#include <iostream>
#include <climits>
```

```
#include <vector>
using namespace std;

int optimalBST(int N, vector<int>& F) {
    // Initialize a 2D dp array with size (N+1)x(N+1)
    vector<vector<int>> dp(N + 2, vector<int>(N + 1, 0));

    // Cumulative sum array
    vector<int> prefixSum(N + 1, 0);
    for (int i = 1; i <= N; ++i) {
        prefixSum[i] = prefixSum[i - 1] + F[i];
    }

    // Fill the dp table
    for (int len = 1; len <= N; ++len) {
        for (int i = 1; i <= N - len + 1; ++i) {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;
            int sum = prefixSum[j] - prefixSum[i - 1];
            for (int k = i; k <= j; ++k) {
                int cost = dp[i][k - 1] + dp[k + 1][j] + sum;
                if (cost < dp[i][j]) {
                    dp[i][j] = cost;
                }
            }
        }
    }

    return dp[1][N];
}

int main() {
    int N;
    cin >> N;

    vector<int> F(N + 1);
    for (int i = 1; i <= N; ++i) {
        cin >> F[i];
    }

    int minComparisons = optimalBST(N, F);
    cout << minComparisons << endl;

    return 0;
}
```