1 Knapsack Problem

Algorithm 1: Knapsack

```
Input: W (weight capacity), wt[] (weights), val[] (values), n (number of items)
Output: Maximum value that can be put in a knapsack of capacity W

Initialize K[n+1][W+1] with 0;
for i = 0 to n do
    for w = 0 to W do
        if i = 0 or w = 0 then
            K[i][w] ← 0;
        else if wt[i-1] ≤ w then
            K[i][w] ← max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
        else
            K[i][w] ← K[i-1][w];
        end
    end
end
return K[n][W];
```

2 Coin Change Problem

Algorithm 2: Coin Change

```
Input: coins[] (coin denominations), amount (target amount)
Output: Minimum number of coins needed to make up the amount

Initialize dp[amount+1] with amount+1;
dp[0] ← 0;
for i = 1 to amount do
    for each coin in coins do
        if coin ≤ i then
            dp[i] ← min(dp[i], dp[i - coin] + 1);
        end
    end
end
if dp[amount] ¿ amount then
    return-1;
else
    return dp[amount];
end
```

3 Longest Common Subsequence (LCS)

Algorithm 3: Longest Common Subsequence

```
Input: S1, S2 (two sequences)
Output: Length of LCS and the LCS itself
m ← length(S1), n ← length(S2);
Initialize dp[m+1][n+1] with 0;
for i = 1 to m do
    for j = 1 to n do
        if S1[i-1] = S2[j-1] then
            dp[i][j] ← dp[i-1][j-1] + 1;
        else
            dp[i][j] ← max(dp[i-1][j], dp[i][j-1]);
        end
    end
end
lcs ← empty string;
i ← m, j ← n;
while i ¿ 0 and j ¿ 0 do
    if S1[i-1] = S2[j-1] then
        Append S1[i-1] to front of lcs;
        i ← i - 1;
        j ← j - 1;
    else if dp[i-1][j] ¿ dp[i][j-1] then
        i ← i - 1;
    else
        j ← j - 1;
    end
end
return (dp[m][n], lcs);
```

4 Karatsuba Integer Multiplication

Algorithm 4: Karatsuba Integer Multiplication

```
Input: x, y (two integers)
Output: Product of x and y
if x ¡ 10 or y ¡ 10 then
    return x * y;
n ← max(number of digits in x, number of digits in y);
m ← n / 2;
a ← left half of x;
b ← right half of x;
c ← left half of y;
d ← right half of y;
ac ← Karatsuba(a, c);
bd ← Karatsuba(b, d);
ad_bc ← Karatsuba(a + b, c + d) - ac - bd;
return ac * 10^(2m) + ad_bc * 10^m + bd;
```

5 Matrix Multiplication

Algorithm 5: Matrix Multiplication

```
Input: A[n][n], B[n][n] (two n x n matrices)
Output: C[n][n] (product of A and B)

Initialize C[n][n] with 0;
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
            C[i][j] ← C[i][j] + A[i][k] * B[k][j];
        end
    end
end
return C;
```

6 Closest Pair Problem

Algorithm 6: Closest Pair

```
Input: P (set of points)
Output: Smallest distance between any two points in P

Sort P by x-coordinate;
return ClosestUtil(P, —P—);

Function ClosestUtil(P, n):
    if n ≤ 3 then
        return BruteForce(P, n);
    mid ← n / 2;
    midPoint ← P[mid];
    dl ← ClosestUtil(P[1..mid], mid);
    dr ← ClosestUtil(P[mid+1..n], n - mid);
    d ← min(dl, dr);
    strip[] ← points in P whose x-distance from midPoint ¡ d;
    return min(d, StripClosest(strip, —strip—, d));

Function StripClosest(strip[], size, d):
    Sort strip by y-coordinate;
    for i = 1 to size do
        j ← i + 1;
        while j ≤ size and (strip[j].y - strip[i].y) ¡ d do
            d ← min(d, dist(strip[i], strip[j]));
            j ← j + 1;
        end
    end
    return d;
```

7 Maxima Set Problem

Algorithm 7: Maxima Set

Input: S (set of points)
Output: Maxima set of S

```
if —S— ≤ 1 then
 | return S;
Sort S by x-coordinate;
mid ← —S— / 2;
p ← S[mid];
L ← {point ∈ S — point ¡ p};
G ← {point ∈ S — point ≥ p};
M1 ← FindMaximaSet(L);
M2 ← FindMaximaSet(G);
q ← point in M2 with minimum x-coordinate;
M1 ← M1 - {r ∈ M1 — r.x ≤ q.x and r.y ≤ q.y};
return M1 ∪ M2;
```