**Coin-row problem**

There is a row of *n* coins whose values are some positive integers but they aren't necessarily distinct. How do we pick up the maximum amount of money when we cannot pick up two coins that are next to each other (no two adjacent coins can be picked up).

Coin row problem algorithm implements dynamic program. The data structure used to implement the coin row problem algorithm is an array.

Algorithm Pseudo-Code

*CoinRow*(C[1…n])

//Input: Array C[1..n] of positive integers indicating coin values

//Output: The maximum amount of money that can be picked up.

F[0] ← 0; F[1] ← C[1]

**for** i ← 2 **to** n **do**

F[i] ← max(C[i] + F[i-2], F[i-1])

**Return** F[n]

**Time complexity: Ɵ(n).** Basic operation of this algorithm is the determination of the maximum using recurrence relation on each iteration for loop.

**Robot coin collection**

There is a *n*x*m* board of cells. Each cell may or may not contain a coin. How do we pick up the maximum amount of coins when we can only move one cell to the right or one cell down on each move?

This problem implements dynamic problem. The data structure used to implement the algorithm would be a multi-dimensional array.

Algorithm Pseudo-Code

*RobotCoinCollection*(C[1..n, 1..m])

//Input: An *n* x *m* matrix whose values or either 1 or 0

//Output: Maximum number of coins that can collect

F[1,1] = C[1,1];

**for** j ← 2 **to** m **do**

F[1, j] ← F[1, j-1]+C[1, j]

**for** i ← 2 **to** n **do**

F[i,1] ← F[i-1, 1] + C[*i*,1]

**for** j ← 2 **to** m **do**

F[i, j] ← max(F[i-1, j], F[i,j-1] + C[i,j])

**return** F[*n*, *m*]

**Knapsack problem**

Given *n* items of known weights *w*1, . . . , *w*n and values *v*1, . . . , *v*n and a knapsack capacity *W* find the most valuable subset of the items that fit into the knapsack. We assume here that all the weights and the knapsack capacity are positive integers; the item values do not have to be integers.

The following recurrence solves this problem:

With the initial conditions:

and

**ALGORITHM** *MFKnapsack(i, j)*

*//*Input: A nonnegative integer *i*  indicating the number of the first

// items being considered and a nonnegative integer *j* indicating the

// knapsack capacity

//Output: The value of an optimal feasible subset of the first *i* items

**if** F[i,j]<0

**if** j < Weights[i]

*value* ← *MFKnapsack*(*i*-1, *j*)

**else**

*value ←* max(*MFKnapsack(i*-1, *j),*

*Values*[*i*] + *MFKnapsack*(*i*-1, *j-Weights[i]))*

*F*[*i*,*j*] ← *value*

**return** *F*[*i*, *j*]

**Prim’s Algorithm**

Start at a vertex, look for minimum edge attached, look at new vertices edges and pick lowest weight edge connected to either vertex until all vertices are connected without a cycle

**ALGORITHM** *Prim(G)*

*//*Input a weighted connected graph G=(V,E)

//Output: ET, the set of edges composing a minimum spanning tree of G

VT ← {V0}

ET ← Ø

**for** i ← 1 **to** |V| - 1 **do**

find a minimum-weight edge e\* = (v\*, u\*) among all the edges (v, u) such that v is in VT and u is in V-VT

VT ← VT ᴜ {u\*}

ET ← ET ᴜ {e\* }

**return** ET

**Kruskal's algorithm**

Collects all minimum edges such that all vertices are attached without a cycle

**ALGORITHM** *Kruskal(G)*

//Input: A weighted connected graph G = <V, E>

//Output: ET, the set of edges composing a minimum spanning tree of G

sort E in nondecreasing order of the edge weights

ET ← Ø; *ecounter ← 0*

*k ← 0*

**while** ecounter < |V| - 1 **do**

k ← k + 1

**if** ET  is acyclic

ET ← ET ; ecounter ← ecounter + 1

**return** ET

**Horspool’s String Matching**

The algorithm is explained thusly:

**Step 1**: for a given search pattern of length *m* and the alphabet used in the pattern and text, construct a shift table.

**Step 2**: Align the pattern against the beginning of the text.

**Step 3**: Repeat the following until either a matching substring is found or the pattern reaches beyond the last letter of the text. Starting with the last letter in the pattern, compare the corresponding characters in the pattern and text until either all *m* characters are matched (then stop) or a mismatching pair is encountered. If a mismatch occurs, determine the letter in the text (call this C) that caused the mismatch. Look up the shift number associated with that letter in the shift table, and shift the pattern by that shift number to the right along the text.