N Queen Problem

1) Start in the leftmost column

2) If all queens are placed

return true

3) Try all rows in the current column. Do following for every tried row.

a) If the queen can be placed safely in this row then mark this [row,

column] as part of the solution and recursively check if placing queen here leads to a solution.

b) If placing the queen in [row, column] leads to a solution then return

true.

c) If placing queen doesn't lead to a solution then umark this [row,

column] (Backtrack) and go to step (a) to try other rows.

3) If all rows have been tried and nothing worked, return false to trigger

backtracking.

Algorithm: Traveling-Salesman-Problem

C ({1}, 1) = 0

for s = 2 to n do

for all subsets S $\in$ {1, 2, 3, … , n} of size s and containing 1

C (S, 1) = ∞

for all j $\in$ S and j $\neq$ 1

C (S, j) = min {C (S – {j}, i) + d(i, j) for i $\in$ S and i $\neq$ j}

Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

Matrix-chain Multiplication

The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices. The standard algorithm is given by the following pseudocode. The attributes *rows* and *columns* are the numbers of rows and columns in a matrix.

```
MATRIX-MULTIPLY(A,B)
1 if columns [A] ≠ rows [B]
2   then error "incompatible dimensions"
3   else for i ←1 to rows [A]
4         do for j ←1 to columns[B]
5               do C[i, j] ← 0
6                  for k ← 1 to columns [A]
7                      do C[i ,j] ← C[i ,j] + A [i, k] B[k, j]
8       return C
```

Bellman ford algorithm

Input: Weighted, undirected graph G=(V,E) with weight function l.
Output: A list {d(v[j]) : j = 1,..,n} containing the distances dist(v[1],v[j]) = d(v[j]), if there are no negative circles reachable from v[1].

The message "Negative Circle" is shown, if a negative circle can be reached from v[1].

```
BEGIN
  d(v[1]) ← 0
  FOR j = 2,..,n DO
    d(v[j]) ← ∞
  FOR i = 1,..,(|V|-1) DO
    FOR ALL (u,v) in E DO
      d(v) ← min(d(v), d(u) + l(u,v))
  FOR ALL (u,v) in E DO
    IF d(v) > d(u) + l(u,v) DO
      Message: "Negative Circle"
END
```

Union-Find

```
def find(data, i):
    if i != data[i]:
        data[i] = find(data, data[i])
    return data[i]
def union(data, i, j):
    pi, pj = find(data, i), find(data, j)
    if pi != pj:
        data[pi] = pj
def connected(data, i, j):
    return find(data, i) == find(data, j)
```

KNUTH-MORRIS-PRATT (T, P)

Input: Strings $T[0 \ldots n]$ and $P[0 \ldots m]$
Output: Starting index of substring of T matching P

```
f ← compute failure function of Pattern P
i ← 0
j ← 0
while i < length[T] do
    if j ← m-1 then
        return i- m+1    // we have a match
i ← i +1
j ← j +1
else if j > 0
        j ← f(j -1)
    else
        i ← i +1
```

Binary Search Algorithm

Binary Search is applied on the sorted array or list of large size. It's time complexity of **O(log n)** makes it very fast as compared to other sorting algorithms. The only limitation is that the array or list of elements must be sorted for the binary search algorithm to work on it.

Implementing Binary Search Algorithm

Following are the steps of implementation that we will be following:

1. Start with the middle element:
    - If the **target** value is equal to the middle element of the array, then return the index of the middle element.
    - If not, then compare the middle element with the target value,
        - If the target value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1.
        - If the target value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1.
2. When a match is found, return the index of the element matched.
3. If no match is found, then return -1

Prim's algorithm

(1) Create an empty tree, M, which will act as the MST.

(2) Choose a random vertex v from the graph.

(3) Add the edges that are connected to v into some data structure E.

(4) Find the edge in E with the minimum weight, and add this edge to M. Now, make v equal to the other vertex in the edge and repeat from step 3.

Kruskal's Algorithm

Algorithm

MST-KRUSKAL(G,w)

1. A = ∅
2. for each vertex v ∈ G.V
3. MAKE-SET(v)
4. sort the edges of G.E into nondecreasing order by weight w
5. for each edge (u,v) ∈ G.E, taken in nondecreasing order by weight
6. if FIND-SET(u) ≠ FIND-SET(v)
7. A = A ∪ {(u,v)}
8. UNION(u,v)
9. return A

Basically the algorithm works as follows:

1. Make each vertex a separate tree
2. Sort the edges in nondecreasing order
3. Add an edge if it connects different trees and merge the trees together

Dynamic-0-1-knapsack (v, w, n, W)

The algorithm takes the following inputs

- The max$_i$mum weight **W**
- The number of items **n**
- The two sequences **v = <v_1, v_2, …, v_n>** and **w = <w_1, w_2, …, w_n>**

for w = 0 to W do

c[0, w] = 0

for i = 1 to n do

```
c[i, 0] = 0
for w = 1 to W do
  if wi ≤ w then
    if vi + c[i-1, w-wi] then
      c[i, w] = vi + c[i-1, w-wi]
    else c[i, w] = c[i-1, w]
  else
    c[i, w] = c[i-1, w]
```

Algorithm HEAPSORT

HEAPSORT (A, N):

An array A with N elements is given.

This algorithm sorts the element of A.

[Build a heap A ,using a procedure 1]

Repeat for J=1 to N-1

Call INSHEAP(A, J, A[J+1])

[sort A by repeatedly deleting the root of H, using procedure 2]

Repeat while N>1:

Call DELHEAP(A , N,VALUE)

Set A[n+1]:=value

Exit