Exhaustive Search

Saravana Senthilkumar - 3122247001057

January 18, 2026

Contents

1 1. Backtracking for Exhaustive Enumeration

1.1 1. Implement Solve(Y, k) to print all permutations of set {0. .n-1}

```
def Solve(Y, k):
```

```
    n = len(Y)
    if k == n:
        print(Y[:k])
        return

    for choice in range(n):
        if choice not in Y[:k]:
            Y[k] = choice
            Solve(Y, k + 1)

a = [None] * 3
Solve(a, 0)
```

1.2 Implement a Python iterator Permutations(n) that returns all permutations of set {0. .n-1}.

1.3 Call the iterator to print all the permutations.

```
def Permutations(n):
    Y = [None] * n
    Solve(Y, 0)
Permutations(4)
```

2 Backtracking for Exhaustive Search

2.1 Formulate the 4-queen problem

```
def Safe(y, r, c):
    for r1 in range(r):
        c1 = y[r1]
        if c1 == c or abs(c - c1) == abs(r - r1):
            return False
    return True


def PlaceQueens(y, j):
    n = len(y)
    if j == n:
        print(y[:])
        return
```

```
    for k in range(n):
        if Safe(y, j, k):
            y[j] = k
            PlaceQueens(y, j + 1)

y = [0] * 4
PlaceQueens(y, 0)
print(y)
```

2.2 Use exhaustive search to find any one safe configuration

```
def IsQueenSafe(y):
    n = len(y)
    for i in range(n):
        for j in range(i + 1, n):
            if abs(y[i] - y[j]) == abs(i - j):
                return False
    return True

def Permute(y, k, n):
    if k == n:
        if IsQueenSafe(y):
            return y[:]
        return None

    for c in range(n):
        if c not in y[:k]:
            y[k] = c
            z = Permute(y, k + 1, n)
            if z is not None:
                return z

    return None
result = Permute([None] * 4, 0, 4)
print(result)
```

2.3 Find all the safe configurations

```
def PermuteAll(y, k, n, solutions):
```

```
    if k == n:
        if IsQueenSafe(y):
            solutions += [y]
        return

    for c in range(n):
        if c not in y[:k]:
            y[k] = c
            PermuteAll(y, k + 1, n, solutions)
y = [None] * 4
PermuteAll(y, 0, 4,[])
print(solutions)
```

2.4 Develop an algorithm PlaceQueens() directly, which prunes the subtree below a partial configuration

```
def PlaceQueens(y, r, solutions):
    n = len(y)
    if r == n:
        solutions += [y]
        return

    for c in range(n):
        if Safe(y, r, c):
            y[r] = c
            PlaceQueens(y, r + 1, solutions)
PlaceQueens([None] * 4, 0, [])
print(solutions)
```

3 Backtracking for Optimization

3.1 Write a function exhaustive enumereate () that enumerates all the subsets of a set {0 . . n 1} of size.

```
def exhaustive_enumerate(x, k, n):
    if k == n:
        print(x)
        return
```

```
    x[k] = 0
    exhaustive_enumerate(x, k + 1, n)

    x[k] = 1
    exhaustive_enumerate(x, k + 1, n)

x = [0] * 3
exhaustive_enumerate(x, 0, 3)
```

3.2 Develop an algorithm to find and print the subsets whose weights add upto W.

```
def SubsetSum(w, x, k, n, W, current_sum):
    if k == n:
        if current_sum == W:
            print(x)
        return


    x[k] = 0
    SubsetSum(w, x, k + 1, n, W, current_sum)


    x[k] = 1
    SubsetSum(w, x, k + 1, n, W, current_sum + w[k])


w = [8, 6, 7, 5, 3]
W = 15
x = [0] * len(w)

SubsetSum(w, x, 0, len(w), W, 0)
```

3.3 Compute the maximum-weight subset not exceeding capacity W

```
def MaxSubsetSum(w, x, k, n, W, current_sum, best):
    if current_sum > W:
        return best
```

```
    if k == n:
        if current_sum > best:
            best = current_sum
        return best


    best = MaxSubsetSum(w, x, k + 1, n, W, current_sum, best)


    x[k] = 1
    best = MaxSubsetSum(w, x, k + 1, n, W, current_sum + w[k], best)
    x[k] = 0

    return best


w = [11, 6, 5, 1, 7, 13, 12]
W = 15
x = [0] * len(w)

best = MaxSubsetSum(w, x, 0, len(w), W, 0, 0)
print(best)
```