Algorísmica Avançada, 2021-2022

Mario Vilar

January 13, 2022

[ ]:
```
def fib_top_down(n, dp=None):
    if not dp: dp = [0]*(n+1)
    if (n == 0) or (n == 1): return n
    if dp[n] != 0: return dp[n]
    dp[n] = fib_top_down(n-1,dp) + fib_top_down(n-2,dp)
    return dp[n]
```

[ ]:
```
def fib_bottom_up(n):
    dp = [0, 1] + [0]*(n-1)
    for i in range(2,n+1): dp[i] = dp[i-1]+dp[i-2]
    return dp[n]
```

[ ]:
```
def rod_cutting_dp_top_down(N, prices, dp=None):
    if not dp: dp = [0]*(N+1)
    if (N == 0): return N
    if (N == 1): return prices[N-1]
    if dp[N] != 0: return dp[N]
    dp[N] = max([prices[N-i-1]+rod_cutting_dp_top_down(i,prices,dp) for i in␣
 ↪range(0,N)])
    return dp[N]
```

[ ]:
```
def cent_savings_rec(lst, d):
    if d == 0: return myround(sum(lst))
    if len(lst) == 0: return 0
    dp = [0]*len(lst)
    for i in range(len(lst)):
        dp[i] = myround(sum(lst[:i]))+cent_savings_rec(lst[i:],d-1)
    return min(dp)
```

[ ]:
```
def cent_savings_dp_bottom_up(lst, d):
    n = len(lst); m = d; arr = [[0]*(m+1) for i in range(n+1)]
    for j in range(m+1): arr[0][j] = 0
    for i in range(1,n+1): arr[i][0] = arr[i-1][0]+lst[i-1]
    for i in range(1,n+1):
        for j in range(m,0,-1):
            arr[i][j] = min(arr[i-1][j]+lst[i-1],myround(arr[i-1][j-1]+lst[i-1]))
    return min([myround(arr[n][j]) for j in range(m+1)])
```

[ ]:
```
def knapsack(W, weights, values, n):
    K = [[0 for x in range(W+1)]for x in range (n+1)]
    for i in range(n+1):
        for w in range(W+1):
```

```
            if(i==0) or (w==0): K[i][w] = 0
            elif weights[i-1] <= w:
                K[i][w] = max(values[i-1] + K[i-1][w-weights[i-1]], K[i-1][w])
            else: K[i][w] = K[i-1][w]
    return K[n][W]
```

[ ]:
```
from itertools import combinations
from collections import defaultdict

def travelling_salesman(G):
    adjMatrix = nx.adjacency_matrix(G)
    n = len(G.nodes())
    memo = defaultdict(lambda: float('inf'))
    for x in range(1, n):
        memo[x , ()] = adjMatrix[0,x]
    for size in range(1, n):
        for k in range(1,n):
            for S in combinations(range(1,n),size):
                if k in S: continue
                for j in S:
                    tup = tuple([ i for i in S if i!=j ])
                    memo[k,S] = min(memo[k,S], adjMatrix[j,k] + memo[j,tup])
    return min([memo[k,tuple([ i for i in range(1,n) if i!=k])] +
                adjMatrix[k,0] for k in range(1,n)])
```

[ ]:
```
def train_sorting(values):
    n = len(values); lds = [1]*n
    for i in range(1,n):
        for j in range(i):
            if values[i] < values[j]:
                lds[i] = max(lds[i],lds[j]+1)
    return max(lds)
```

[ ]:
```
def train_sorting(values):
    n = len(values)
    dp = [[0]*(n+1) for i in range(n+1)]
    for i in range(1,n+1):
        dp[i][0] = values[i-1]; dp[0][i] = values[i-1]
        dp[i][i] = 1
    for length in range(2,n+1):
        for i in range(1,n+2-length):
            j = i+length-1
            if values[i-1] < values[j-1]:
                dp[i][j] = dp[i+1][j-1] + 1
            else:
                dp[i][j] = max(dp[i][j-1], dp[i+1][j])
    return dp[1][n]
```

[ ]:
```
def longest_palindrom_subsequence_dp(seq):
    n = len(seq)
    dp = [[0]*(n+1) for i in range(n+1)]
    for i in range(1,n+1):
        dp[i][i] = 1
```

```python
    for length in range(2,n+1):
        for i in range(1,n+2-length):
            j = i+length-1
            if seq[i-1] == seq[j-1]:
                dp[i][j] = dp[i+1][j-1] + 2
            else:
                dp[i][j] = max(dp[i][j-1], dp[i+1][j])
    return dp
```

[ ]:
```python
def solve_deck_backtracking(N, solution, placed_nums):
    if 0 not in solution: return True
    for i in range(N,0,-1):
        lst = valid_movement(i,N,placed_nums,solution)
        if lst:
            solution[lst[0]],solution[lst[1]] = i, i
            placed_nums.add(i)
            if(solve_deck_backtracking(N,solution,placed_nums)):
                return solution
            solution[lst[0]],solution[lst[1]] = 0, 0
            placed_nums.remove(i)
    return False
```

[ ]:
```python
def sum_K_backtracking(lst, K, tmp_sum, idx, sub_list):
    if tmp_sum == K: return True
    for i in range(idx,len(lst)):
        if tmp_sum + lst[i] <= K:
            tmp_sum += lst[i] # place movement
            sub_list.append(lst[i]) # place movement
            if sum_K_backtracking(lst, K, tmp_sum, idx+1, sub_list):
                print(sub_list)
            tmp_sum -= lst[i] # unplace movement
            sub_list.remove(lst[i]) # unplace movement
    return False
```

[ ]:
```python
def solve_queens(N):
    board = [[0]*N for _ in range(N)]
    solution = solve_queens_backtracking(N, board, 0)

    if not solution: return f'N={N}: No té solució'
    return f'N={N}:\n{format_board(solution)}'

def solve_queens_backtracking(N, board, col):
    if col == N: return True
    for i in range(len(board)):
        if(check_position_previous_columns(board, i, col)):
            board[i][col] = 1
            if(solve_queens_backtracking(N, board, col+1)):
                return board
            board[i][col] = 0
    return False
```

[ ]:
```python
def inf_bound(matrix):
    return sum(matrix.min(axis=0)) if len(matrix)!=0 else 0
```

```python
def sup_bound(matrix):
    return sum(matrix.diagonal()) if len(matrix)!=0 else 0

def tasks(matrix):
    sup = sup_bound(matrix)
    inf = inf_bound(matrix)
    pq = PriorityQueue()
    pq.put((inf, [], 0, set([])))
    while not pq.empty():
        elem_cota, elem_list, elem_row, elem_cols = pq.get()
        for col in range(len(matrix)):
            if col not in elem_cols:
                new_elem_list, new_elem_cols = elem_list.copy(), elem_cols.copy()
                new_elem_cols.add(col)
                new_elem_list.append((elem_row, col))
                new_elem_row = elem_row + 1
                if len(new_elem_list) == len(matrix)-1:
                    erow, ecol = len(matrix)-1, list(set(range(len(matrix))) -␣
 ↪new_elem_cols)[0]
                    new_elem_cols.add(ecol); new_elem_list.append((erow, ecol))
                    new_elem_row += 1
                matrix_slice = np.delete(matrix, list(range(0,new_elem_row)), 0)
                matrix_slice = np.delete(matrix_slice, list(new_elem_cols), 1)
                new_elem_cota = sum(matrix[i,j] for i,j in new_elem_list) +␣
 ↪inf_bound(matrix_slice)
                if len(new_elem_list) == len(matrix):
                    if new_elem_cota < sup:
                        sup = new_elem_cota
                elif new_elem_cota < sup:
                    pq.put((new_elem_cota, new_elem_list, new_elem_row, new_elem_cols))
```

[ ]:

```python
from queue import PriorityQueue
import numpy as np
def solve_puzzle(board):
    best_bound = np.inf; best_board = board; pq = PriorityQueue()
    pq.put((board.manhattan_distance(), 0, board))
    existent_states = set([board.get_state_id()]); expanded = 0
    while not pq.empty():
        inf_bound, current_bound, board = pq.get(); expanded += 1
        for i in board.allowed_moves():
            new_board = board.move(i)
            if new_board.get_state_id() not in existent_states:
                new_bound = current_bound + new_board.manhattan_distance()
                if(new_bound > best_bound):
                    continue
                elif(new_board.state()):
                    if new_bound < best_bound:
                        best_board = new_board; best_bound = new_bound
                else:
                    existent_states.add(new_board.get_state_id())
                    pq.put((new_board.manhattan_distance(), new_bound, new_board))
    return (best_bound, best_board, expanded)
```