Algorithms and Data Structure 10

Yiping Deng

May 7, 2018

Problem 1

```python
# Yiping Deng
def main_func():
    # main function to take the input

    # take the input
    inp = list(map(int, input().split())) #convert it into list

    rst = longestOrderedSubarray(inp)

    print(*rst, sep=' ')

def longestOrderedSubarray(arr):
    """
    Calculate the longest subarray
    >>> longestOrderedSubarray([8,3,6,50,10,8,100,30,60,40,80])
    [3, 6, 10, 30, 60, 80]
    """
    cache = {} #memoization
    def helper(start, previous):
        """
        Calculate the longest ordered subarray.
        arr: list of integers
        start: index of the start of the array
        end: index of the end of the array
        previous: the biggest element in the previous subarray

        return the array
        """
        nonlocal cache

        if start >= len(arr):
            return []

        # if not yet computed
        if (start, previous) not in cache:

            # if you cannot take this element
            if arr[start] < previous:
                cache[(start, previous)] = helper(start + 1, previous)
            else:
                # if you can take it, find the best case
                inclusive = helper(start + 1, arr[start])
                exclusive = helper(start + 1, previous)

                # consider two cases, and decide the best case
                if (len(inclusive) + 1) >= len(exclusive):
                    cache[(start, previous)] = [arr[start]] + inclusive
                else:
                    cache[(start, previous)] =  exclusive

        return cache[(start, previous)]

    # call the helper function
    # use -infinity so that you can always take the first
    # one
    return helper(0, float('-inf'))

if __name__ == '__main__':
    main_func()
```

Problem 2

```
def main_func():
    inp = []

    while True:
        # continously taking input
        inp_line = list(map(int, input().split()))
        if len(inp_line) == 0:
            break
        inp.append(inp_line)

    #calculate the result
    total_sum, path = decideroad(inp)

    #print the result
    print(total_sum)
    print(*path, sep = ' ')

def decideroad(arr):
    cache = {}
    def helper(level, position):
        # helper function decide road at different level
        nonlocal cache
        if level >= len(arr):
            # out of bound
            return []
        if (level, position) not in cache:
            if level == 0:
                cache[(level, position)] = [arr[level][position]] + helper(1, 0)
            else:
                left = [arr[level][position]] + helper(level + 1, position)
                right = [arr[level][position]] + helper(level + 1, position + 1)

                leftSum = sum(left)
                rightSum = sum(right)

                if leftSum >= rightSum:
                    cache[(level, position)] = left
                else:
                    cache[(level, position)] = right
        return cache[(level, position)]
    rst = helper(0,0)
    return sum(rst), rst

if __name__ == '__main__':
    main_func()
```

Problem 3

```
def main_func():
    # do some reading and execute the strategy
    num_cases = int(input()) # take the input

    inputs = []
    for _ in range(num_cases):

        target_line = input().split()

        oxygen = int(target_line[0])
        nitrogen = int(target_line[1])

        num_cylinders = int(input())

        cylinders = []

        for _ in range(num_cylinders):
            cylinder_line = input().split()

            cylinder_oxy = int(cylinder_line[0])
            cylinder_nitro = int(cylinder_line[1])
            cylinder_weight = int(cylinder_line[2])

            cylinders.append((cylinder_oxy, cylinder_nitro, cylinder_weight))

        inputs.append((oxygen, nitrogen, cylinders))

    # start solving the case
    results = map(lambda tup: scubaSolve(tup[2], tup[0], tup[1]), inputs)

    for result_total, result_cylinders in results:
        print(result_total)
        one_indexed = map(lambda x: x + 1, result_cylinders)
        print(*one_indexed, sep = ' ')


def scubaSolve(cylinders, oxygen, nitrogen):
    # such cache is actually 3 dimension, so let it be
    cache = {} # dynamic programming

    def helper(idx, oxygen, nitrogen):
        nonlocal cache # access the dynamic programming cache

        if oxygen <= 0 and nitrogen <= 0:
            # solution reached, return 0 and empty list
            return 0, []

        if idx >= len(cylinders):
            # no solution given
            return None

        if (idx, oxygen, nitrogen) not in cache:
            cylinder_oxy, cylinder_nitro, cylinder_weight = cylinders[idx]

            # break into two cases
            # included is guaratee to have solution
            included = helper(idx + 1, oxygen - cylinder_oxy, nitrogen - cylinder_nitro)

            if included is None:
                # if not solvable
                cache[(idx, oxygen, nitrogen)] = None
            else:
                # default optimal
                excluded = helper(idx + 1, oxygen, nitrogen)

                optimal_total, optimal_cylinders = included
                optimal_total, optimal_cylinders = optimal_total + cylinder_weight, [idx] + optimal_

                if excluded is not None:
                    # find the optimal solution
                    excluded_total, excluded_cylinders = excluded
                    if optimal_total > excluded_total:
                        optimal_total, optimal_cylinders = excluded_total, excluded_cylinders

                # put it into the cache
                cache[(idx, oxygen, nitrogen)] = optimal_total, optimal_cylinders

        return cache[(idx, oxygen, nitrogen)]

    return helper(0, oxygen, nitrogen)

if __name__ == '__main__':
    main_func()
```