[Python] Multiple recursive and dp solutions

Quantum73 ★ 22 September 20, 2021 5:34 AM 147 VIEWS

3

Solution 1: Recursive solution with 2 options in each state. Either add or subtract element in `stones`. Use caching: O(sum*L).

```
def lastStoneWeightII(self, stones: List[int]) -> int:
        @lru_cache(None)
        def dfs(csum, i):
            if i < 0 and csum >= 0: return csum
            if i < 0: return float('inf')
            return min(dfs(csum+stones[i], i-1), dfs(csum-stones[i], i-1))
        return dfs(0, len(stones)-1)
```

Solution 2: Find subsequence with sum closest (and smaller) to `sum(stones)/2`.

```
def lastStoneWeightII(self, stones: List[int]) -> int:
        @lru_cache(None)
        def dfs(diff, i):
            if diff < 0: return -float('inf')
            if i < 0 or diff == 0: return 0
            return max(stones[i] + dfs(diff - stones[i], i-1), dfs(diff, i-1))
        S = sum(stones)
        return S - int(2*dfs(S/2, len(stones)-1))
```

Solution 3: (alternative to solution 2). Find subsequence with sum closest (and smaller) to `sum(stones)/2`.

```
def lastStoneWeightII(self, stones: List[int]) -> int:
        # O(sum*len(stones)) time and space
        @lru_cache(None)
        def dfs(diff, i):
            if diff < 0: return float('inf')
```

Solution 3: (alternative to solution 2). Find subsequence with sum closest (and smaller) to `sum(stones)/2`.

```
def lastStoneWeightII(self, stones: List[int]) -> int:
        # O(sum*len(stones)) time and space
        @lru_cache(None)
        def dfs(diff, i):
            if diff < 0: return float('inf')
            if i < 0 or diff == 0: return diff
            return min(dfs(diff - stones[i], i-1), dfs(diff, i-1))
        return int(2*dfs(sum(stones)/2, len(stones)-1))
```

Solution 4: DP. Find subsequence with sum closest (and smaller) to `sum(stones)/2`. Similar to knapsack 0-1.

```
def lastStoneWeightII(self, stones: List[int]) -> int:
        # DP: O(sum*len(stones)) time and space
        L = len(stones)
        S = sum(stones)
        dp = [[0]*(S//2+1) for _ in range(L+1)]
def lastStoneWeightII(self, stones: List[int]) -> int:
        # DP: O(sum*len(stones)) time and space
        L = len(stones)
        S = sum(stones)
        dp = [[0]*(S//2+1) for _ in range(L+1)]
        for i in range(1, L+1):
            for j in range(1, S//2+1):
                if stones[i-1] <= j:
                    dp[i][j] = max(dp[i-1][j], stones[i-1]+dp[i-1][j-stones[i-1]])
                else:
                    dp[i][j] = dp[i-1][j]
        return S -2*dp[L][-1]
```

Solution 5: DP but with O(sum) in space.

```
def lastStoneWeightII(self, stones: List[int]) -> int:
        # DP: O(sum*len(stones)) time and O(sum) space
        S = sum(stones)
        dp = [0]*(S//2+1)
        for i in range(len(stones)):
            for j in range(S//2, stones[i]-1, -1):
                    dp[j] = max(dp[j], stones[i]+dp[j-stones[i]])
        return S-2*dp[-1]
```

knapsack dynamic programming recursion memoization