<https://leetcode.com/problems/minimum-cost-to-merge-stones/description/>

There are n piles of stones arranged in a row. The ith pile has stones[i] stones.

A move consists of merging exactly k **consecutive** piles into one pile, and the cost of this move is equal to the total number of stones in these k piles.

Return *the minimum cost to merge all piles of stones into one pile*. If it is impossible, return -1.

**Example 1:**

Input: stones = [3,2,4,1], k = 2

Output: 20

Explanation: We start with [3, 2, 4, 1].

We merge [3, 2] for a cost of 5, and we are left with [5, 4, 1].

We merge [4, 1] for a cost of 5, and we are left with [5, 5].

We merge [5, 5] for a cost of 10, and we are left with [10].

The total cost was 20, and this is the minimum possible.

**Example 2:**

Input: stones = [3,2,4,1], k = 3

Output: -1

Explanation: After any merge operation, there are 2 piles left, and we can't merge anymore. So the task is impossible.

**Example 3:**

Input: stones = [3,5,1,2,6], k = 3

Output: 25

Explanation: We start with [3, 5, 1, 2, 6].

We merge [5, 1, 2] for a cost of 8, and we are left with [3, 8, 6].

We merge [3, 8, 6] for a cost of 17, and we are left with [17].

The total cost was 25, and this is the minimum possible.

**Constraints:**

* n == stones.length
* 1 <= n <= 30
* 1 <= stones[i] <= 100
* 2 <= k <= 30

**Attempt 1: 2023-09-04**

**Solution 1: Native DFS (360 min, TLE 42/84)**

class Solution {

public int mergeStones(int[] stones, int k) {

int n = stones.length;

// e.g stones = [3,2,4,1], k = 3 -> expect -1

if((n - 1) % (k - 1) != 0) {

return -1;

}

// Calculating the prefix sum so that sum of any segment[i..j] can be calculated easily

int[] preSum = new int[n + 1];

for(int i = 1; i <= n; i++) {

preSum[i] = preSum[i - 1] + stones[i - 1];

}

// Find the minimum cost to merge stones[0...n-1] into 1 pile

return helper(0, n - 1, 1, preSum, k);

}

// How to use 'preSum' ?

// e.g

// stones = [1,2,3,4,5]

// preSum = [0,1,3,6,10,15]

// i = 1, j = 4 -> expect 2 + 3 + 4 + 5 = 14

// preSum[j + 1] - preSum[i] = 15 - 1 = 14

// i = 0, j = 3 -> expect 1 + 2 + 3 + 4 = 10

// preSum[j + 1] - preSum[i] = 10 - 0 = 10

private int helper(int i, int j, int piles, int[] preSum, int k) {

// Base condition 1:

// Cost of converting segment[i...i] into 1 pile is 0

if(i == j && piles == 1) {

return 0;

}

// Base condition 2:

// Cost of converting segment[i..i] into other than 1

// pile is not possible, so placed MAX value

if(i == j) {

// Divide INT.MAX by 4 then no overflow on integer

return Integer.MAX\_VALUE / 4;

}

// If segment[i...j] is converted into 1 pile

if(piles == 1) {

// dp(i,j,1) = dp(i,j,k) + sum[i...j]

return helper(i, j, k, preSum, k) + preSum[j + 1] - preSum[i];

} else {

// dp(i,j,piles) = min(dp(i,j,piles), dp(i,t,1) + dp(t+1,j,piles-1)) for all t E i<=t<j

int cost = Integer.MAX\_VALUE / 4;

for(int t = i; t < j; t++) {

cost = Math.min(cost, helper(i, t, 1, preSum, k) + helper(t + 1, j, piles - 1, preSum, k));

}

return cost;

}

}

}

**Solution 2:  DFS + Memoization (10 min)**

class Solution {

public int mergeStones(int[] stones, int k) {

int n = stones.length;

// e.g stones = [3,2,4,1], k = 3 -> expect -1

if((n - 1) % (k - 1) != 0) {

return -1;

}

// memo[i][j][k] means the minimum cost to merge

// array from index i to j into k piles

Integer[][][] memo = new Integer[n][n][k + 1];

// Calculating the prefix sum so that sum of any segment[i..j] can be calculated easily

int[] preSum = new int[n + 1];

for(int i = 1; i <= n; i++) {

preSum[i] = preSum[i - 1] + stones[i - 1];

}

// Find the minimum cost to merge stones[0...n-1] into 1 pile

return helper(0, n - 1, 1, preSum, k, memo);

}

// How to use 'preSum' ?

// e.g

// stones = [1,2,3,4,5]

// preSum = [0,1,3,6,10,15]

// i = 1, j = 4 -> expect 2 + 3 + 4 + 5 = 14

// preSum[j + 1] - preSum[i] = 15 - 1 = 14

// i = 0, j = 3 -> expect 1 + 2 + 3 + 4 = 10

// preSum[j + 1] - preSum[i] = 10 - 0 = 10

private int helper(int i, int j, int piles, int[] preSum, int k, Integer[][][] memo) {

// Base condition 1:

// Cost of converting segment[i...i] into 1 pile is 0

if(i == j && piles == 1) {

return 0;

}

// Base condition 2:

// Cost of converting segment[i..i] into other than 1

// pile is not possible, so placed MAX value

if(i == j) {

// Divide INT.MAX by 4 then no overflow on integer

return Integer.MAX\_VALUE / 4;

}

if(memo[i][j][piles] != null) {

return memo[i][j][piles];

}

// If segment[i...j] is converted into 1 pile

if(piles == 1) {

// dp(i,j,1) = dp(i,j,k) + sum[i...j]

return helper(i, j, k, preSum, k, memo) + preSum[j + 1] - preSum[i];

} else {

// dp(i,j,piles) = min(dp(i,j,piles), dp(i,t,1) + dp(t+1,j,piles-1)) for all t E i<=t<j

int cost = Integer.MAX\_VALUE / 4;

for(int t = i; t < j; t++) {

cost = Math.min(cost, helper(i, t, 1, preSum, k, memo) + helper(t + 1, j, piles - 1, preSum, k, memo));

}

return memo[i][j][piles] = cost;

}

}

}

**Explained to make you Visualize the solution . Detailed Explanation**

**Refer to**

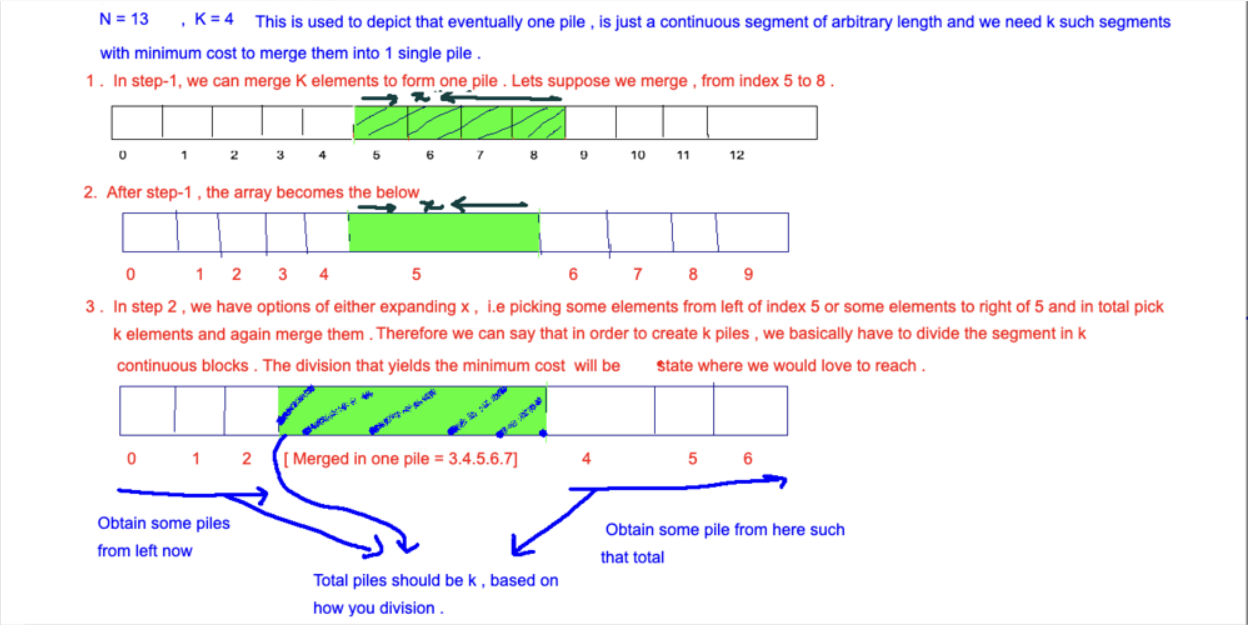
<https://leetcode.com/problems/minimum-cost-to-merge-stones/solutions/1432667/explained-to-make-you-visualise-the-solution-detailed-explanation/>

This problem is amongst one of the **harder problems** in dynamic programming . It took me too long to understand the the logic behind the problem and I usually failed to visualize the solution mentioned in other posts . However @cappachino post inspired me to write my own post of how I visualized his explanation and put it in public for people failing to understand the solution .

##### **Here is my attempt to actually explain what I understood and how I visualize the problem**

1. We know that after all merge operations , we will be left with only **one pile**.
2. Now lets begin thinking , that in order to actually create one pile at the end , we in the previous step should be left with K piles so as to merge them into one pile . **However in order to obtain the minimum cost of creating one pile , we must end up creating K piles in previous step with minimum cost.**
3. Now the question arises how do we create K piles in the previous steps such that those costs minimum cost . To understand this , we must understand what does a pile represents . **A pile here is basically a continuous segment in the array without any restriction on the length** . How does we arrive to the definition of a pile ? .

##### **Below is an attempt definition of a pile .**



1. Now based on above figure , not sure I was successful in explaining the above , but what I meant to explain was that when you are merging a consecutive segment of k piles , some of these piles could have been from already merged segment (say x shown in digram ) , and we eventually are increasing the length of this segment if we involve x in other merging operations which simply means the pile is same but its length is increased . And at the end , we just require k piles with minimum cost .
2. If my attempt to make you understand what a pile is successful , then the above problem reduces to actually dividing the array into k segments such that the total cost for reaching to this k piles in minimum .
3. Let's define the state for the the problem :

dp(i,j,k) = means the minimum cost to merge array from index i to index j into k different piles

1. Now , since state is known , here are transitions between the states

Transitions

============

dp(i,j,1) = dp(i,j,k) + sum[i..j]

where sum[i..j] represents the sum between index i and index j .

Which means that in order to create one pile from index i to

to index j (dp(i,j,1)) in minimum cost , we have to create k piles

from index i to index j (dp(i,j,k)) and merge the operation cost

which is sum of the segment.

dp(i,j,k) = dp(i,t,1) + dp(t+1,j,k-1) where t lies between index i to j

where i is inclusive and j is exclusive .

which means that in order to create k pile from index i to

index j , we first choose any segment of arbitary length

and try creating the pile from (i,t) and then check for the

minimum cost of creating (k-1) piles from the rest of the

array .

Base Cases :

==============

dp(i,i,1) = Since only merge operation has cost therfore , and we dont need

merge in the interval i to i to create 1 pile, therefore cost is 0 .

1. Now , if you have understood the above , please go through the code . I have added comments to support my above explanation .

#include<bits/stdc++.h>

using namespace std;

class Solution {

public:

int dp[50][50][50];

int minCost(int i,int j,int piles,vector<int>&prefixsum,int &K){

// Cost of converting segment [i..i] into 1 pile is zero

if( i == j && piles == 1)

return 0;

// Cost of converting segment[i..i] into other than 1 pile is not possible , so placed MAX value

if(i == j)

return INT\_MAX/4;

// Check whether the subproblem has already been solved .

if(dp[i][j][piles]!=-1)

return dp[i][j][piles];

// If segment[i...j] is to converted into 1 pile

if(piles == 1){

// Here dp(i,j,1) = dp(i,j,K) + sum[i...j]

return dp[i][j][piles] = minCost(i,j,K,prefixsum,K) + (i==0 ? prefixsum[j] : prefixsum[j]-prefixsum[i-1]);

} else {

// dp(i,j,piles) = min( dp(i,j,piles), dp(i,t,1) + dp(t+1,j,piles-1)) for all t E i<=t<j

int cost = INT\_MAX/4;

for(int t=i;t<j;t++){

cost = min(cost, minCost(i,t,1,prefixsum,K) + minCost(t+1,j,piles-1,prefixsum,K));

}

return dp[i][j][piles] = cost;

}

}

int mergeStones(vector<int>& stones, int k) {

// the size of the stones

int n = stones.size();

/\*

Check whether we will be able to merge n piles into 1 pile .

In step-1 we merge k pile and then we are left with n-k+1 piles or n-(k-1);

In Step-2 we again merge k pile and then we are left with ((n-k+1)-k)+1 or n-2\*(k-1);

In Step-3 we gain merge k pile and left with (((n-k+1)-k+1)-k)+1 or n-3\*(k-1)

.......

.......

.......

After some m steps we should be left with 1 pile

Therefore , n-m\*(k-1) == 1

(n-1)-m\*(k-1)=0;

Since m needs to be an integer therefore ,

if (n-1)%(k-1) == 0 ,

then we can merge otherwise not possible.

\*/

if((n-1)%(k-1)!=0)

return -1;

int sum = 0;

vector<int>prefixsum;

// Calculating the prefix sum so that sum of any segment[i..j] can be calculated easily

for(int i=0;i<stones.size();i++){

sum+=stones[i];

prefixsum.push\_back(sum);

}

memset(dp,-1,sizeof(dp));

// find the minimum cost to merge stones[0...n-1] into 1 pile

return minCost(0,n-1,1,prefixsum,k);

}

};

**Solution 3:  DP (360 min)**

class Solution {

public int mergeStones(int[] stones, int k) {

int n = stones.length;

// e.g stones = [3,2,4,1], k = 3 -> expect -1

if((n - 1) % (k - 1) != 0) {

return -1;

}

// Calculating the prefix sum so that sum of any segment[i..j] can be calculated easily

int[] preSum = new int[n + 1];

for(int i = 1; i <= n; i++) {

preSum[i] = preSum[i - 1] + stones[i - 1];

}

int[][] dp = new int[n][n];

// Find the minimum cost to merge stones[0...n-1] into 1 pile

for(int len = k; len <= n; len++) {

for(int i = 0; i + len <= n; i++) {

int j = i + len - 1;

dp[i][j] = Integer.MAX\_VALUE;

for(int t = i; t < j; t += k - 1) {

dp[i][j] = Math.min(dp[i][j], dp[i][t] + dp[t + 1][j]);

}

if((j - i) % (k - 1) == 0) {

dp[i][j] += preSum[j + 1] - preSum[i];

}

}

}

return dp[0][n - 1];

}

}

**Refer to**

<https://grandyang.com/leetcode/1000/>

这道题给了我们N堆石头，每堆石头有不同的个数，说每次可以合并K堆石头，合并堆的花费就是石头的个数，然后问如何合并，才能使总花费最小。然后给了一些例子，通过观察例子，可以发现，并不是所有的输入都能成功合成一堆，比如例子2，无论先和并哪三堆，最终都会剩下两堆，从而无法进一步合并，因为 K=3，每次至少需要合并三堆。我们当然希望能在开始合并之前就能知道最终是否能成功合并为一堆，而不是算到最后了才发现白忙了一场，所以要来分析一下，什么时候才能最终合并为一堆。再来看看例子2，每次要将三堆合并为一堆，那么就是减少了两堆，而要使得最终能够剩下一堆，其他的都要合并调，假设原来共有n堆，只能剩下一堆，就是说 n-1 堆都要减掉，而每次只能减少 k-1 堆，所以只要 n-1 能够整除 k-1即可，即 (n-1)%(k-1) == 0 成立，这样就可以提前判断了。

好，接下来继续，考虑如何来解题，首先要意识到这道题的情况可能非常多，用暴力搜索的话可能会非常的复杂，而且当前的合并方法完全会影响到之后的合并，所以基本是要放弃 Brute force 的想法的。同样，这道题也不能用贪婪算法，每次都合并石子个数最少的三堆会收敛到局部峰值，不一定是全局的，所以只能另辟蹊径。观察到这题是玩数组的，又是求极值的题目，那么就要祭出神器动态规划 Dynamic Programming 了，先来考虑定义 dp 数组吧，最简单直接的方法肯定直接用个二维的dp数组了，其中 dp[i][j] 表示合并范围 [i, j] 内的石头堆的最小花费，最终 dp[0][n-1] 就是所要求的值。看到了论坛上有人定义了三维的 dp 数组，把每次合并的堆数K也当作一维放入到 dp 数组中了，其实博主觉得不是很有必要，因为像这种必须要对 dp 数组进行升维操作的是当题目中有隐藏信息 Hidden Information，而当前定义的 dp 数组无法重现子问题，即无法找到状态转移方程的时候必须要做的，最典型的例子就是之前那道 [Remove Boxes](http://www.cnblogs.com/grandyang/p/6850657.html)，那道题自区间的 dp 值非常依赖于区间左边相同的数字的个数，而这道题每次合并的堆数K并不是很依赖其他小于K的合并的堆数，所以博主感觉没有必要加。关于含有隐藏信息的 dp 题目，感觉巅峰就属于拣樱桃那题 [Cherry Pickup](http://www.cnblogs.com/grandyang/p/8215787.html) 了吧，现在看还是有点晕，改天还得重新加工一下吧。其实跟这道题最像的当属于打气球那题 [Burst Balloons](https://www.cnblogs.com/grandyang/p/5006441.html)，气球爆了之后，两边的气球就挨到一起了，这里也很类似，石子合并之后，再跟两边的石子堆继续合并，这里的更新方式还是从小区间更新到大区间，跟打气球那题的思路非常的相似，建议先去看看博主的之前那篇博客 [Burst Balloons](https://www.cnblogs.com/grandyang/p/5006441.html)，会对理解这道题大有裨益。

根据之前打气球的经验，要从小区间开始更新，多小呢，从K开始，因为小于K的区间不用更新，其 dp 值一定为0，因为每次必须合并K堆石子，所以区间的长度 len 从K遍历到 n。好，区间长度确定了，现在要确定起点了，i从0遍历到 n-len 即可，有了区间的起点和长度，可以确定区间的终点 j = i+len-1。目标就是要更新区间 [i, j] 的dp值，先初始化为整型最大值。接下来的更新方法，即状态转移方程，就是本题最大的难点了，要求区间 [i, j] 的 dp 值，没法直接得到，但是由于是从小区间开始更新的，所以 suppose 其中的小区间的 dp 值都已经更新好了，就可以将大区间拆成两个小区间来更新了。一般来讲，将一个数组拆成两个非空子数组的时候，会遍历其所有情况，比如 [1, 2, 3, 4]，会拆成 [1] 和 [2,3,4]，[1,2] 和 [3,4], [1,2,3] 和 [4]。但是这道题由于其特殊性，并不需要遍历所有的拆分情况，因为某些区间是无法通过合并石子堆得到的，就拿上面的例子来说，若 K=3，那么就不需要用 [1,2] 和 [3,4] 来更新整个区间，它们都不到3个，无法合并，所以遍历的时候每次跳过 K-1 个位置即可，用 t 来分别区间 [i, j]，然后每次 t += K-1 即可，用两个小区间的 dp 值来更新整个区间。对于这两个小区间的dp值并不能直接加权得出，因为这两个小区间的元素数量不是K的整数倍，不满足直接加权的要求，但这两个小区间的dp也不能直接舍弃，因为它们本身可能内部包含了K的整数倍元素个数的区间，后面总会有需要计算出结果的时候。这还没有完，当某个子区间正好可以合并为一堆石子的时候，其 dp 值要加上该区间所有的石子数。举个最简单的例子，比如 [1, 2, 3]，K=3，那么我们分割的话，只能用 dp[0][0] + dp[1][2] 来更新 dp[0][2]，但是 dp[0][0] 和 dp[1][2] 均为0，因为区间长度均小于3，那么我们的 dp[0][2] 值就无法更新成正确的值了，这三个数字是可以合并的，所以要加上区间内所有的数字之和，而为了快速的求得任意区间和，采用提前建立累加和数组 sums 的方式，来提高计算效率，所以整个状态转移方程为：

dp[i][j] = min(dp[i][j], dp[i][t] + dp[t + 1][j]); -> (i <= t < j)

dp[i][j] += sums[j + 1] - sums[i]; -> if ((j - i) % (K - 1) == 0)

有了状态转移方程，我们就可以写出代码如下：

class Solution {

public:

int mergeStones(vector<int>& stones, int K) {

int n = stones.size();

if ((n - 1) % (K - 1) != 0) return -1;

vector<int> sums(n + 1);

vector<vector<int>> dp(n, vector<int>(n));

for (int i = 1; i < n + 1; ++i) {

sums[i] = sums[i - 1] + stones[i - 1];

}

for (int len = K; len <= n; ++len) {

for (int i = 0; i + len <= n; ++i) {

int j = i + len - 1;

dp[i][j] = INT\_MAX;

for (int t = i; t < j; t += K - 1) {

dp[i][j] = min(dp[i][j], dp[i][t] + dp[t + 1][j]);

}

if ((j - i) % (K - 1) == 0) {

dp[i][j] += sums[j + 1] - sums[i];

}

}

}

return dp[0][n - 1];

}

};