【华为OD机考 统一考试机试C卷】伐木工（C++ Java JavaScript Python C语言）

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷

目前在考C卷，经过两个月的收集整理，C卷真题已基本整理完毕
抽到原题的概率为2/3到3/3，也就是最少抽到两道原题。请注意：大家刷完C卷真题，最好要把B卷的真题刷一下，因为C卷的部分真题来自B卷。
另外订阅专栏还可以联系笔者开通在线 OJ 进行刷题，提高刷题效率。
真题目录：华为OD机考机试 真题目录（C卷 + D卷 + B卷 + A卷） + 考点说明
专栏：2023华为OD机试(B卷+C卷+D卷)（C++JavaJSPy）
华为OD面试真题精选：华为OD面试真题精选
在线OJ：点击立即刷题，模拟真实机考环境

题目描述

一根X米长的树木，伐木工切割成不同长度的木材后进行交易，交易价格为每根木头长度的乘积。规定切割后的每根木头长度都为正整数；也可以不切割，直接拿整根树木进行交易。

请问伐木工如何尽量少的切割，才能使收益最大化?

输入描述

木材的长度（X ≤ 50）

输出描述

输出最优收益时的各个树木长度，以空格分隔，按升序排列

用例

输入

```
10
```

输出

```
3 3 4
```

说明

> 一根2米长的树木，伐木工不切割，为2 * 1，收益最大为2
> 一根4米长的树木，伐木工不需要切割为2 * 2，省去切割成本，直接整根树木交易，为4 * 1，收益最大为4
> 一根5米长的树木，伐木工切割为2 * 3，收益最大为6
> 一根10米长的树木，伐木工可以切割方式一：3，4，4，也可以切割为方式二：3，2，2，3，但方式二伐木工多切割一次，增加切割成本却买了一样的价格，因此并不是最优收益。

解题思路

C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
// 定义函数，计算给定长度的最大乘积分割
vector<int> getMaxProfit(int length) {
    // dp数组用于存储每个长度的最大乘积
    vector<int> dp(length + 1);
    // cutTimes数组用于存储每个长度的最佳切割次数
    vector<int> cutTimes(length + 1);
    // lastCut数组用于存储每个长度的最后一次切割长度
    vector<int> lastCut(length + 1);

    // 遍历每个长度
    for (int i = 1; i <= length; ++i) {
        // 初始化dp和lastCut数组
        dp[i] = lastCut[i] = i;
        // 遍历所有可能的切割长度
        for (int j = 1; j < i; ++j) {
            // 计算当前切割长度的乘积
            int product = dp[i - j] * j;
            // 如果当前乘积大于已知的最大乘积，更新最大乘积和最佳切割长度
            if (product > dp[i]) {
```

```
                lastCut[i] = j;
                dp[i] = product;
                cutTimes[i] = cutTimes[i - j] + 1;
            }
            // 如果当前乘积等于已知的最大乘积，但切割次数更少，更新最佳切割长度和切割次数
            else if (product == dp[i] && cutTimes[i] > cutTimes[i - j] + 1) {
                lastCut[i] = j;
                cutTimes[i] = cutTimes[i - j] + 1;
            }
        }
    }

    // 创建一个vector来存储结果
    vector<int> results;
    // 从最大长度开始，每次减去最佳切割长度，直到长度为0
    while (length > 0) {
        // 将最佳切割长度添加到结果的开头
        results.insert(results.begin(), lastCut[length]);
        // 更新长度
        length -= lastCut[length];
    }
    // 对结果进行排序
    sort(results.begin(), results.end());
    // 返回结果
    return results;
}

int main() {
    // 读取输入的长度
    int length;
    cin >> length;
    // 调用getMaxProfit方法计算最大利润，并将结果存储在一个vector中
    vector<int> results = getMaxProfit(length);
    // 遍历结果并打印
    for (int i : results) {
        cout << i << " ";
    }
    return 0;
}
```

Java

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        // 创建一个扫描器来读取用户输入
        Scanner scanner = new Scanner(System.in);
        // 读取输入的长度
        int length = scanner.nextInt();
        // 调用getMaxProfit方法计算最大利润，并将结果存储在一个ArrayList中
        ArrayList<Integer> results = getMaxProfit(length);
        // 遍历结果并打印
        for (int i : results) {
            System.out.print(i + " ");
        }
    }

    private static ArrayList<Integer> getMaxProfit(int length) {
        // dp数组用于存储每个长度的最大乘积
        int[] dp = new int[length + 1];
        // cutTimes数组用于存储每个长度的最佳切割次数
        int[] cutTimes = new int[length + 1];
        // lastCut数组用于存储每个长度的最后一次切割长度
        int[] lastCut = new int[length + 1];

        // 遍历每个长度
        for (int i = 1; i <= length; i++) {
            // 初始化dp和lastCut数组
            dp[i] = lastCut[i] = i;
            // 遍历所有可能的切割长度
            for (int j = 1; j < i; j++) {
                // 计算当前切割长度的乘积
                int product = dp[i - j] * j;
                // 如果当前乘积大于已知的最大乘积，更新最大乘积和最佳切割长度
                if (product > dp[i]) {
                    lastCut[i] = j;

```

```
                    dp[i] = product;
                    cutTimes[i] = cutTimes[i - j] + 1;
                }
                // 如果当前乘积等于已知的最大乘积，但切割次数更少，更新最佳切割长度和切割次数
                else if (product == dp[i] && cutTimes[i] > cutTimes[i - j] + 1) {
                    lastCut[i] = j;
                    cutTimes[i] = cutTimes[i - j] + 1;
                }
            }
        }

        // 创建一个ArrayList来存储结果
        ArrayList<Integer> results = new ArrayList<>();
        // 从最大长度开始，每次减去最佳切割长度，直到长度为0
        while (length > 0) {
            // 将最佳切割长度添加到结果的开头
            results.add(0, lastCut[length]);
            // 更新长度
            length -= lastCut[length];
        }
        // 对结果进行排序
        Collections.sort(results);

        // 返回结果
        return results;
    }
}
```

javaScript

```
// 引入readline模块用于读取输入
const readline = require('readline');

// 创建readline接口
const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

// 定义函数getMaxProfit，计算给定长度的最大乘积分割
```

```
const getMaxProfit = (length) => {
    // dp数组用于存储每个长度的最大乘积
    const dp = new Array(length + 1).fill(0);
    // cutTimes数组用于存储每个长度的最佳切割次数
    const cutTimes = new Array(length + 1).fill(0);
    // lastCut数组用于存储每个长度的最后一次切割长度
    const lastCut = new Array(length + 1).fill(0);

    // 遍历每个长度，从1到length
    for (let i = 1; i <= length; i++) {
        // 初始化dp和lastCut数组
        dp[i] = lastCut[i] = i;
        // 遍历所有可能的切割长度，从1到i
        for (let j = 1; j < i; j++) {
            // 计算当前切割长度的乘积
            const product = dp[i - j] * j;
            // 如果当前乘积大于已知的最大乘积，更新最大乘积和最佳切割长度
            if (product > dp[i]) {
                dp[i] = product;
                lastCut[i] = j;
                cutTimes[i] = cutTimes[i - j] + 1;
            }
            // 如果当前乘积等于已知的最大乘积，但切割次数更少，更新最佳切割长度和切割次数
            else if (product === dp[i] && cutTimes[i] > cutTimes[i - j] + 1) {
                lastCut[i] = j;
                cutTimes[i] = cutTimes[i - j] + 1;
            }
        }
    }

    // 创建一个数组来存储结果
    const results = [];
    // 从最大长度开始，每次减去最佳切割长度，直到长度为0
    while (length > 0) {
        // 将最佳切割长度添加到结果的开头
        results.unshift(lastCut[length]);
        // 更新长度
        length -= lastCut[length];
    }
    // 对结果进行排序
```

```
    results.sort((a, b) => a - b);
    // 返回结果
    return results;
};

rl.on('line', (input) => {
    // 将输入转换为整数
    const length = parseInt(input);
    // 调用getMaxProfit方法计算最大利润，并将结果存储在一个数组中
    const results = getMaxProfit(length);
    // 打印结果
    console.log(results.join(' '));
    // 关闭readline接口
    rl.close();
});
```

Python

```
# 定义函数，计算给定长度的最大乘积分割
def get_max_profit(length):
    # dp数组用于存储每个长度的最大乘积
    dp = [0] * (length + 1)
    # cutTimes数组用于存储每个长度的最佳切割次数
    cut_times = [0] * (length + 1)
    # lastCut数组用于存储每个长度的最后一次切割长度
    last_cut = [0] * (length + 1)

    # 遍历每个长度
    for i in range(1, length + 1):
        # 初始化dp和lastCut数组
        dp[i] = last_cut[i] = i
        # 遍历所有可能的切割长度
        for j in range(1, i):
            # 计算当前切割长度的乘积
            product = dp[i - j] * j
            # 如果当前乘积大于已知的最大乘积，更新最大乘积和最佳切割长度
            if product > dp[i]:
                last_cut[i] = j
                dp[i] = product
                cut_times[i] = cut_times[i - j] + 1
```

```
            # 如果当前乘积等于已知的最大乘积，但切割次数更少，更新最佳切割长度和切割次数
            elif product == dp[i] and cut_times[i] > cut_times[i - j] + 1:
                last_cut[i] = j
                cut_times[i] = cut_times[i - j] + 1

    # 创建一个列表来存储结果
    results = []
    # 从最大长度开始，每次减去最佳切割长度，直到长度为0
    while length > 0:
        # 将最佳切割长度添加到结果的开头
        results.insert(0, last_cut[length])
        # 更新长度
        length -= last_cut[length]
    # 对结果进行排序
    results.sort()
    # 返回结果
    return results

# 读取输入的长度
length = int(input(""))
# 调用get_max_profit方法计算最大利润，并将结果存储在一个列表中
results = get_max_profit(length)
# 打印结果
print(' '.join(map(str, results)))
```

C语言

```
#include <stdio.h>
#include <stdlib.h>

// 定义函数，计算给定长度的最大乘积分割
void getMaxProfit(int length, int* results, int* resultsSize) {
    // dp数组用于存储每个长度的最大乘积
    int* dp = (int*)malloc((length + 1) * sizeof(int));
    // cutTimes数组用于存储每个长度的最佳切割次数
    int* cutTimes = (int*)malloc((length + 1) * sizeof(int));
    // lastCut数组用于存储每个长度的最后一次切割长度
    int* lastCut = (int*)malloc((length + 1) * sizeof(int));

    // 遍历每个长度
```

```
    for (int i = 1; i <= length; ++i) {
        // 初始化dp和lastCut数组
        dp[i] = lastCut[i] = i;
        // 遍历所有可能的切割长度
        for (int j = 1; j < i; ++j) {
            // 计算当前切割长度的乘积
            int product = dp[i - j] * j;
            // 如果当前乘积大于已知的最大乘积，更新最大乘积和最佳切割长度
            if (product > dp[i]) {
                lastCut[i] = j;
                dp[i] = product;
                cutTimes[i] = cutTimes[i - j] + 1;
            }
            // 如果当前乘积等于已知的最大乘积，但切割次数更少，更新最佳切割长度和切割次数
            else if (product == dp[i] && cutTimes[i] > cutTimes[i - j] + 1) {
                lastCut[i] = j;
                cutTimes[i] = cutTimes[i - j] + 1;
            }
        }
    }

    // 从最大长度开始，每次减去最佳切割长度，直到长度为0
    while (length > 0) {
        // 将最佳切割长度添加到结果的开头
        results[*resultsSize] = lastCut[length];
        (*resultsSize)++;
        // 更新长度
        length -= lastCut[length];
    }

    // 对结果进行排序
    for (int i = 0; i < *resultsSize; i++) {
        for (int j = i + 1; j < *resultsSize; j++) {
            if (results[i] > results[j]) {
                int temp = results[i];
                results[i] = results[j];
                results[j] = temp;
            }
        }
    }
```

```
    // 释放动态分配的内存
    free(dp);
    free(cutTimes);
    free(lastCut);
}

int main() {
    // 读取输入的长度
    int length;
    scanf("%d", &length);

    // 创建一个数组来存储结果
    int* results = (int*)malloc(length * sizeof(int));
    int resultsSize = 0;

    // 调用getMaxProfit方法计算最大利润，并将结果存储在一个数组中
    getMaxProfit(length, results, &resultsSize);

    // 遍历结果并打印
    for (int i = 0; i < resultsSize; i++) {
        printf("%d ", results[i]);
    }

    // 释放动态分配的内存
    free(results);

    return 0;
}
```

文章目录

C++
Java
javaScript
Python
C语言