# 【华为OD机考 统一考试机试C卷】项目排期（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

项目组共有N个开发人员，项目经理接到了M个独立的需求，每个需求的工作量不同，且每个需求只能由一个开发人员独立完成，不能多人合作。假定各个需求直接无任何先后依赖关系，请设计算法帮助项目经理进行工作安排，使整个项目能用最少的时间交付。

## 输入描述

第一行输入为M个需求的工作量，单位为天，用逗号隔开。

> 例如：X1 X2 X3 … Xm 。表示共有M个需求，每个需求的工作量分别为X1天，X2天…Xm天。
> 其中0<M<30；0<Xm<200

第二行输入为项目组人员数量N

## 输出描述

最快完成所有工作的天数

## 用例

输入：

```
6 2 7 7 9 3 2 1 3 11 4
2
```

输出：

```
28
```

说明：

> 共有两位员工，其中一位分配需求 6 2 7 7 3 2 1共需要28天完成，另一位分配需求 9 3 11 4 共需要27天完成，故完成所有工作至少需要28天。

## 解题思路

给定一系列任务的工作量和一定数量的工人，计算完成所有任务所需的最少天数，使得每个工人分配到的任务总工作量不超过这个天数。这是一个典型的搜索问题，可以通过回溯法和二分查找结合来解决。

1. **排序和反转任务数组**：
   - 使用 `Arrays.sort(tasks)` 对任务数组进行升序排序，然后通过一个循环将数组反转，使其成为降序。这样做是为了优先分配工作量大的任务，从而更高效地利用工人的工作时间。
2. **二分查找**：
   - 为了找到完成所有任务所需的最少天数，使用二分查找确定这个最小值。设置两个指针 `l` 和 `r` ，分别表示可能的最短时间的下界和上界。 `l` 初始化为数组中的最大值（即最大的单个任务工作量）， `r` 初始化为所有任务工作量的总和。
   - 在 `l` 小于 `r` 的条件下进行循环，计算中间值 `mid` ，并使用 `canFinish` 函数检查是否可以在 `mid` 天内完成所有任务。
   - 如果可以完成，则将上界 `r` 设置为 `mid` ，否则将下界 `l` 设置为 `mid + 1` 。
   - 当 `l` 和 `r` 相遇时， `l` 即为所求的最少天数。
3. **回溯法**：
   - `canFinish` 函数使用回溯法来检查在给定的时间限制 `limit` 内是否可以完成所有任务。
   - 创建一个长度为工人数量 `k` 的数组 `workers` ，用于记录每个工人的当前工作量。
   - 使用 `backtrack` 函数递归地尝试为每个任务分配工人，直到所有任务都被分配或者无法在时间限制内完成分配。

- 在 backtrack 函数中，如果当前工人可以在时间限制内完成当前任务，则将任务分配给他，并递归地尝试分配下一个任务。
- 如果分配成功，则返回 true ； 如果当前路径无法成功分配所有任务，则回溯到上一个状态，尝试其他可能的分配方案。
- 如果所有方案都无法成功，则返回 false 。

## C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <sstream>
using namespace std;
// 回溯法
bool backtrack(vector<int>& tasks, vector<int>& workers, int index, int limit) {
    // 如果所有任务都已分配，则返回true
    if (index >= tasks.size()) {
        return true;
    }

    // 获取当前任务的工作量
    int current = tasks[index];
    // 尝试将当前任务分配给每个员工
    for (int i = 0; i < workers.size(); i++) {
        // 如果当前员工可以在时间限制内完成这项任务
        if (workers[i] + current <= limit) {
            // 分配任务给当前员工
            workers[i] += current;
            // 继续尝试分配下一个任务
            if (backtrack(tasks, workers, index + 1, limit)) {
                return true;
            }
            // 回溯，取消当前的任务分配
            workers[i] -= current;
        }

        // 如果当前员工没有任务或者加上当前任务刚好达到时间限制，则不需要尝试其他员工
        if (workers[i] == 0 || workers[i] + current == limit) {
            break;
        }
```

```
    }

    // 如果无法分配当前任务，则返回false
    return false;
}
// 检查是否可以在给定的时间限制内完成所有任务
bool canFinish(vector<int>& tasks, int k, int limit) {
    // 创建一个数组来记录每个员工的工作量
    vector<int> workers(k, 0);
    // 使用回溯法检查是否可以完成
    return backtrack(tasks, workers, 0, limit);
}
// 计算完成所有任务所需的最少天数
int minimumTimeRequired(vector<int>& tasks, int k) {
    // 将任务按工作量降序排序
    sort(tasks.begin(), tasks.end(), greater<int>());

    // 使用二分查找确定完成所有任务的最短时间
    int l = tasks[0], r = accumulate(tasks.begin(), tasks.end(), 0);
    while (l < r) {
        int mid = (l + r) / 2;
        // 检查当前时间限制是否足够完成所有任务
        if (canFinish(tasks, k, mid)) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }

    // 返回最短完成时间
    return l;
}




int main() {
    // 使用cin读取输入
    vector<int> tasks;
    string input;
```

```
    getline(cin, input);
    istringstream iss(input);
    int value;
    while (iss >> value) {
        tasks.push_back(value);
    }
    int N;
    cin >> N;

    // 输出最快完成所有工作的天数
    cout << minimumTimeRequired(tasks, N) << endl;
    return 0;
}
```

## Java

```
import java.util.Arrays;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        // 使用Scanner读取输入
        Scanner scanner = new Scanner(System.in);
        // 读取第一行输入，即需求的工作量，并以空格分隔
        String[] workloads = scanner.nextLine().split(" ");
        // 读取第二行输入，即项目组人员数量
        int N = Integer.parseInt(scanner.nextLine());
        // 创建一个数组来存放每个需求的工作量
        int[] tasks = new int[workloads.length];

        // 将输入的工作量转换为整数并存入数组
        for (int i = 0; i < workloads.length; i++) {
            tasks[i] = Integer.parseInt(workloads[i]);
        }

        // 输出最快完成所有工作的天数
        System.out.println(minimumTimeRequired(tasks, N));
    }

    // 计算完成所有任务所需的最少天数
```

```
public static int minimumTimeRequired(int[] tasks, int k) {
    // 将任务按工作量升序排序
    Arrays.sort(tasks);
    // 将排序后的数组反转，使之成为降序
    int low = 0, high = tasks.length - 1;
    while (low < high) {
        int temp = tasks[low];
        tasks[low] = tasks[high];
        tasks[high] = temp;
        low++;
        high--;
    }

    // 使用二分查找确定完成所有任务的最短时间
    int l = tasks[0], r = Arrays.stream(tasks).sum();
    while (l < r) {
        int mid = (l + r) / 2;
        // 检查当前时间限制是否足够完成所有任务
        if (canFinish(tasks, k, mid)) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }

    // 返回最短完成时间
    return l;
}

// 检查是否可以在给定的时间限制内完成所有任务
private static boolean canFinish(int[] tasks, int k, int limit) {
    // 创建一个数组来记录每个员工的工作量
    int[] workers = new int[k];
    // 使用回溯法检查是否可以完成
    return backtrack(tasks, workers, 0, limit);
}

// 回溯法
private static boolean backtrack(int[] tasks, int[] workers, int index, int limit) {
    // 如果所有任务都已分配，则返回true
```

```
        if (index >= tasks.length) {
            return true;
        }

        // 获取当前任务的工作量
        int current = tasks[index];
        // 尝试将当前任务分配给每个员工
        for (int i = 0; i < workers.length; i++) {
            // 如果当前员工可以在时间限制内完成这项任务
            if (workers[i] + current <= limit) {
                // 分配任务给当前员工
                workers[i] += current;
                // 继续尝试分配下一个任务
                if (backtrack(tasks, workers, index + 1, limit)) {
                    return true;
                }
                // 回溯，取消当前的任务分配
                workers[i] -= current;
            }

            // 如果当前员工没有任务或者加上当前任务刚好达到时间限制，则不需要尝试其他员工
            if (workers[i] == 0 || workers[i] + current == limit) {
                break;
            }
        }

        // 如果无法分配当前任务，则返回false
        return false;
    }
}
```

## javaScript

```
const readline = require('readline');

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

```

```
// 读取输入
rl.on('line', (line) => {
  if (!this.tasks) {
    // 第一次输入，处理任务工作量
    this.tasks = line.split(' ').map(Number);
  } else {
    // 第二次输入，处理员工数量
    const N = Number(line);
    // 输出最快完成所有工作的天数
    console.log(minimumTimeRequired(this.tasks, N));
    rl.close();
  }
});

// 计算完成所有任务所需的最少天数
function minimumTimeRequired(tasks, k) {
  // 将任务按工作量降序排序
  tasks.sort((a, b) => b - a);

  // 使用二分查找确定完成所有任务的最短时间
  let l = tasks[0], r = tasks.reduce((a, b) => a + b, 0);
  while (l < r) {
    let mid = Math.floor((l + r) / 2);
    // 检查当前时间限制是否足够完成所有任务
    if (canFinish(tasks, k, mid)) {
      r = mid;
    } else {
      l = mid + 1;
    }
  }

  // 返回最短完成时间
  return l;
}

// 检查是否可以在给定的时间限制内完成所有任务
function canFinish(tasks, k, limit) {
  // 创建一个数组来记录每个员工的工作量
  let workers = new Array(k).fill(0);
  // 使用回溯法检查是否可以完成
```

```
  return backtrack(tasks, workers, 0, limit);
}

// 回溯法
function backtrack(tasks, workers, index, limit) {
  // 如果所有任务都已分配，则返回true
  if (index >= tasks.length) {
    return true;
  }

  // 获取当前任务的工作量
  let current = tasks[index];
  // 尝试将当前任务分配给每个员工
  for (let i = 0; i < workers.length; i++) {
    // 如果当前员工可以在时间限制内完成这项任务
    if (workers[i] + current <= limit) {
      // 分配任务给当前员工
      workers[i] += current;
      // 继续尝试分配下一个任务
      if (backtrack(tasks, workers, index + 1, limit)) {
        return true;
      }
      // 回溯，取消当前的任务分配
      workers[i] -= current;
    }

    // 如果当前员工没有任务或者加上当前任务刚好达到时间限制，则不需要尝试其他员工
    if (workers[i] === 0 || workers[i] + current === limit) {
      break;
    }
  }

  // 如果无法分配当前任务，则返回false
  return false;
}
```

## Python

```
# Python版本代码
from itertools import combinations
```

```python

def minimumTimeRequired(tasks, k):
    # 将任务按工作量降序排序
    tasks.sort(reverse=True)

    # 使用二分查找确定完成所有任务的最短时间
    l, r = tasks[0], sum(tasks)
    while l < r:
        mid = (l + r) // 2
        # 检查当前时间限制是否足够完成所有任务
        if canFinish(tasks, k, mid):
            r = mid
        else:
            l = mid + 1

    # 返回最短完成时间
    return l

def canFinish(tasks, k, limit):
    # 创建一个数组来记录每个员工的工作量
    workers = [0] * k
    # 使用回溯法检查是否可以完成
    return backtrack(tasks, workers, 0, limit)

def backtrack(tasks, workers, index, limit):
    # 如果所有任务都已分配，则返回True
    if index >= len(tasks):
        return True

    # 获取当前任务的工作量
    current = tasks[index]
    # 尝试将当前任务分配给每个员工
    for i in range(len(workers)):
        # 如果当前员工可以在时间限制内完成这项任务
        if workers[i] + current <= limit:
            # 分配任务给当前员工
            workers[i] += current
            # 继续尝试分配下一个任务
            if backtrack(tasks, workers, index + 1, limit):
                return True

```

```
            # 回溯，取消当前的任务分配
            workers[i] -= current

        # 如果当前员工没有任务或者加上当前任务刚好达到时间限制，则不需要尝试其他员工
        if workers[i] == 0 or workers[i] + current == limit:
            break

    # 如果无法分配当前任务，则返回False
    return False

if __name__ == "__main__":
    # 使用input读取输入
    tasks = list(map(int, input().split()))
    N = int(input())

    # 输出最快完成所有工作的天数
    print(minimumTimeRequired(tasks, N))
```

## C语言

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

#define MAX_TASKS 30 // 定义最大任务数量的常量，用于设置任务数组的最大长度

// 用于qsort函数的比较函数，实现降序排序
int compare(const void *a, const void *b) {
    // 将void指针转换为int指针，并解引用获取值进行比较
    return (*(int*)b - *(int*)a);
}

// 回溯法分配任务
int backtrack(int *tasks, int *workers, int index, int limit, int k, int taskSize) {
    // 检查是否所有任务都已分配
    if (index >= taskSize) {
        return 1; // 如果是，返回1表示成功
    }

    // 获取当前要分配的任务
```

```
    int current = tasks[index];
    // 遍历所有员工
    for (int i = 0; i < k; i++) {
        // 检查当前员工是否可以在时间限制内完成这个任务
        if (workers[i] + current <= limit) {
            // 如果可以，分配任务并递归尝试分配下一个任务
            workers[i] += current;
            if (backtrack(tasks, workers, index + 1, limit, k, taskSize)) {
                return 1;
            }
            // 如果不成功，回溯，即撤销这次任务分配
            workers[i] -= current;
        }

        // 如果当前员工没有任务或者加上当前任务刚好达到时间限制，则不需要尝试其他员工
        if (workers[i] == 0 || workers[i] + current == limit) {
            break;
        }
    }

    // 如果无法分配当前任务，返回0表示失败
    return 0;
}

// 检查是否能在指定时间内完成所有任务
int canFinish(int *tasks, int k, int limit, int taskSize) {
    // 初始化一个记录员工当前任务量的数组
    int workers[MAX_TASKS] = {0};
    // 调用回溯法尝试分配任务
    return backtrack(tasks, workers, 0, limit, k, taskSize);
}

// 计算完成所有任务的最短时间
int minimumTimeRequired(int *tasks, int k, int taskSize) {
    // 先对任务进行降序排序
    qsort(tasks, taskSize, sizeof(int), compare);

    // 二分查找的左右边界，左边界为最大单个任务时间，右边界为所有任务时间总和
    int l = tasks[0], r = 0;
    for (int i = 0; i < taskSize; i++) {
```

```
        r += tasks[i];
    }

    // 二分查找最短完成时间
    while (l < r) {
        int mid = l + (r - l) / 2;
        // 检查是否能在mid时间内完成所有任务
        if (canFinish(tasks, k, mid, taskSize)) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }

    // 返回最短完成时间
    return l;
}

int main() {
    // 存储任务的数组和任务数量
    int tasks[MAX_TASKS], taskSize = 0;
    // 读取一行输入作为任务工作量
    char input[200];
    fgets(input, 200, stdin);
    // 使用strtok分割字符串，将分割后的数字转换为int存入任务数组
    char *token = strtok(input, " ");
    while (token != NULL) {
        tasks[taskSize++] = atoi(token);
        token = strtok(NULL, " ");
    }

    // 读取员工数量
    int N;
    scanf("%d", &N);

    // 计算并输出完成所有任务的最短时间
    printf("%d\n", minimumTimeRequired(tasks, N, taskSize));
    return 0;
}
```

**文章目录**