【华为OD机考 统一考试机试C卷】执行任务赚获取最多积分 （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

现有N个任务需要处理，同一时间只能处理一个任务，处理每个任务所需要的时间固定为1。

每个任务都有最晚处理时间限制和积分值，在最晚处理时间点之前处理完成任务才可获得对应的积分奖励。

可用于处理任务的时间有限，请问在有限的时间内，可获得的最多积分。

输入描述

第一行为一个数 N，表示有 N 个任务

- 1 ≤ N ≤ 100

第二行为一个数 T，表示可用于处理任务的时间

- 1 ≤ T ≤ 100

接下来 N 行，每行两个空格分隔的整数（SLA 和 V），SLA 表示任务的最晚处理时间，V 表示任务对应的积分。

- 1 ≤ SLA ≤ 100
- 0 ≤ V ≤ 100000

输出描述

可获得的最多积分

用例1

输入

```
4
3
1 2
1 3
1 4
1 5
```

输出

```
5
```

说明

> 虽然有3个单位的时间用于处理任务，可是所有任务在时刻1之后都无效。
> 所以在第1个时间单位内，选择处理有5个积分的任务。1-3时无任务处理。

用例2

输入

```
4
3
1 2
1 3


```

```
1 4
3 5
```

输出

```
9
```

说明

> 第1个时间单位内，处理任务3，获得4个积分
> 第2个时间单位内，处理任务4，获得5个积分
> 第3个时间单位内，无任务可处理
> 共获得9个积分

解题思路

1. 首先，我们将所有任务按照截止时间进行分类，并在每个截止时间内按照积分从高到低进行排序。这样做的目的是为了在每个截止时间内优先处理积分最高的任务。
2. 然后，我们从最大的截止时间开始，逆序遍历每一个截止时间。这样做的目的是为了优先处理截止时间最晚的任务，因为这些任务有更大的灵活性。
3. 对于每个截止时间，我们将所有的任务添加到剩余的任务列表中，。
4. 如果剩余的任务列表不为空，我们就处理积分最高的任务，并将其积分添加到总积分中并对其进行排序，以便优先处理积分最高的任务。这是贪心选择，我们总是选择当前可用的积分最高的任务。

模拟计算过程：

假设测试用例是：

```
4
3
1 2
2 3
3 4
3 5
```

这个测试用例的意思是，有4个任务，最大时间是3。每个任务的截止时间和积分分别是(1, 2)，(2, 3)，(3, 4)和(3, 5)。

下面是这个测试用例的模拟计算过程：

1. 读取任务数量（4）和最大时间（3）。

   读取每个任务的截止时间和积分，并将它们添加到对应的截止时间列表中。截止时间列表如下：

   - 时间1：[2]
   - 时间2：[3]
   - 时间3：[4, 5]

   从最大时间开始，逆序遍历每一个时间点，将当前时间点的所有任务添加到剩余的任务列表中，并取出积分最高的任务。

   - 时间3：剩余的任务列表为[4, 5]，取出积分最高的任务（5），总积分为5。
   - 时间2：剩余的任务列表为[4, 3]，取出积分最高的任务（4），总积分为9。
   - 时间1：剩余的任务列表为[3, 2]，取出积分最高的任务（3），总积分为12。

4. 输出总积分（12）。

所以，对于这个测试用例，程序的输出应该是12。

C++

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;

int main() {
    // 创建输入流读取输入
    int n, T;
    // 读取任务数量
    cin >> n;
    // 读取可用于处理任务的总时间
    cin >> T;
    // 创建任务向量，每个任务是一个pair，第一个元素代表最晚处理时间，第二个元素代表积分
    vector<pair<int, int>> tasks;

    // 循环读取每个任务的信息
    for (int i = 0; i < n; ++i) {
```

```
        // 读取任务的最晚处理时间
        int deadline;
        // 读取任务的积分
        int score;
        cin >> deadline >> score;
        // 将任务添加到向量中
        tasks.push_back({deadline, score});
    }

    // 按照任务的最晚处理时间升序排序
    sort(tasks.begin(), tasks.end());

    // 创建优先队列（小根堆），用于存储任务的积分
    priority_queue<int, vector<int>, greater<int>> queue;
    // 遍历每个任务
    for (const auto& task : tasks) {
        // 获取任务的最晚处理时间和积分
        int deadline = task.first, score = task.second;
        // 如果当前队列的大小小于任务的最晚处理时间，说明任务还未过期，可以添加到队列中
        if (queue.size() < deadline) {
            queue.push(score);
        }
        // 如果队列不为空，且队列顶部的任务积分小于当前任务的积分
        else if (!queue.empty() && queue.top() < score) {
            // 移除队列顶部的任务
            queue.pop();
            // 将当前任务的积分添加到队列中
            queue.push(score);
        }
        // 如果当前队列的大小已经达到总时间T，说明不可以再处理新的任务
        if (queue.size() > T) {
            // 移除队列顶部的任务
            queue.pop();
        }
    }

    // 初始化总积分为0
    int totalScore = 0;
    // 当队列不为空时，继续处理
    while (!queue.empty()) {

```

```
        // 累加队列顶部的任务积分到总积分
        totalScore += queue.top();
        queue.pop();
    }

    // 输出可以获得的最多积分
    cout << totalScore << endl;
    return 0;
}
```

Java

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // 创建扫描器读取输入
        Scanner scanner = new Scanner(System.in);
        // 读取任务数量
        int n = scanner.nextInt();
        // 读取可用于处理任务的总时间
        int T = scanner.nextInt();
        // 创建任务列表，每个任务是一个包含两个整数的数组，分别代表最晚处理时间和积分
        List<int[]> tasks = new ArrayList<>();

        // 循环读取每个任务的信息
        for (int i = 0; i < n; i++) {
            // 读取任务的最晚处理时间
            int deadline = scanner.nextInt();
            // 读取任务的积分
            int score = scanner.nextInt();
            // 将任务添加到列表中
            tasks.add(new int[]{deadline, score});
        }

        // 按照任务的最晚处理时间升序排序
        tasks.sort(Comparator.comparingInt(a -> a[0]));

        // 创建优先队列，用于存储任务的积分
        PriorityQueue<Integer> queue = new PriorityQueue<>();
```

```
        // 遍历每个任务
        for (int[] task : tasks) {
            // 获取任务的最晚处理时间和积分
            int deadline = task[0], score = task[1];
            // 如果当前队列的大小小于任务的最晚处理时间，说明任务还未过期，可以添加到队列中
            if (queue.size() < deadline) {
                queue.add(score);
            }
            // 如果队列不为空，且队列顶部的任务积分小于当前任务的积分
            else if (!queue.isEmpty() && queue.peek() < score) {
                // 移除队列顶部的任务
                queue.poll();
                // 将当前任务的积分添加到队列中
                queue.add(score);
            }
            // 如果当前队列的大小已经达到总时间T，说明不可以再处理新的任务
            if (queue.size() > T) {
                // 移除队列顶部的任务
                queue.poll();
            }
        }

        // 初始化总积分为0
        int totalScore = 0;
        // 当队列不为空时，继续处理
        while (!queue.isEmpty()) {
            // 累加队列顶部的任务积分到总积分
            totalScore += queue.poll();
        }

        // 输出可以获得的最多积分
        System.out.println(totalScore);
    }
}
```

javaScript

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

// 创建readline接口实例
```

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

// 创建一个小根堆类
class MinHeap {
    constructor() {
        this.heap = [];
    }

    // 获取父节点的索引
    getParentIndex(i) {
        return Math.floor((i - 1) / 2);
    }

    // 获取左子节点的索引
    getLeftChildIndex(i) {
        return 2 * i + 1;
    }

    // 获取右子节点的索引
    getRightChildIndex(i) {
        return 2 * i + 2;
    }

    // 交换堆中两个元素的位置
    swap(i, j) {
        [this.heap[i], this.heap[j]] = [this.heap[j], this.heap[i]];
    }

    // 向堆中插入一个元素
    push(key) {
        this.heap.push(key);
        this.heapifyUp();
    }

    // 移除并返回堆顶元素（最小值）
    pop() {
        if (this.size() === 1) {
```

```
            return this.heap.pop();
        }

        const top = this.heap[0];
        this.heap[0] = this.heap.pop();
        this.heapifyDown();
        return top;
    }

    // 返回堆顶元素（最小值）
    peek() {
        return this.heap[0];
    }

    // 返回堆的大小
    size() {
        return this.heap.length;
    }

    // 向上调整堆
    heapifyUp() {
        let index = this.heap.length - 1;
        while (this.getParentIndex(index) >= 0 && this.heap[this.getParentIndex(index)] > this.heap[index]) {
            this.swap(this.getParentIndex(index), index);
            index = this.getParentIndex(index);
        }
    }

    // 向下调整堆
    heapifyDown() {
        let index = 0;
        while (this.getLeftChildIndex(index) < this.size()) {
            let smallerChildIndex = this.getLeftChildIndex(index);
            if (this.getRightChildIndex(index) < this.size() && this.heap[this.getRightChildIndex(index)] < this.heap[smallerChildIndex]) {
                smallerChildIndex = this.getRightChildIndex(index);
            }

            if (this.heap[index] < this.heap[smallerChildIndex]) {
                break;
            } else {

```

```
                this.swap(index, smallerChildIndex);
            }

            index = smallerChildIndex;
        }
    }
}


let lineCount = 0;
let n, T;
let tasks = [];
let totalScore = 0;

rl.on('line', (line) => {
    if (lineCount === 0) {
        n = parseInt(line);
    } else if (lineCount === 1) {
        T = parseInt(line);
    } else {
        const [deadline, score] = line.split(' ').map(Number);
        tasks.push({ deadline, score });
        if (tasks.length === n) {
            rl.close();
        }
    }
    lineCount++;
});

rl.on('close', () => {
    tasks.sort((a, b) => a.deadline - b.deadline);
    const minHeap = new MinHeap();
    tasks.forEach(task => {
        const { deadline, score } = task;
        if (minHeap.size() < deadline) {
            minHeap.push(score);
        } else if (minHeap.size() > 0 && minHeap.peek() < score) {
            minHeap.pop();
            minHeap.push(score);
        }

```

```
        if (minHeap.size() > T) {
            minHeap.pop();
        }
    });

    while (minHeap.size() > 0) {
        totalScore += minHeap.pop();
    }

    console.log(totalScore);
});
```

Python

```python
import heapq

def main():
    # 读取任务数量和总时间
    n =  int(input())
    T =  int(input())
    # 创建任务列表
    tasks = []

    # 循环读取每个任务的信息
    for _ in range(n):
        deadline, score = map(int, input().split())
        # 将任务添加到列表中
        tasks.append((deadline, score))

    # 按照任务的最晚处理时间升序排序
    tasks.sort(key=lambda x: x[0])

    # 创建优先队列（小根堆），用于存储任务的积分
    queue = []
    # 遍历每个任务
    for deadline, score in tasks:
        # 如果当前队列的大小小于任务的最晚处理时间，说明任务还未过期，可以添加到队列中
        if len(queue) < deadline:
            heapq.heappush(queue, score)
```

```
        # 如果队列不为空，且队列顶部的任务积分小于当前任务的积分
        elif queue and queue[0] < score:
            # 移除队列顶部的任务
            heapq.heappop(queue)
            # 将当前任务的积分添加到队列中
            heapq.heappush(queue, score)
        # 如果当前队列的大小已经达到总时间T，说明不可以再处理新的任务
        if len(queue) > T:
            # 移除队列顶部的任务
            heapq.heappop(queue)

    # 初始化总积分为0
    total_score = 0
    # 当队列不为空时，继续处理
    while queue:
        # 累加队列顶部的任务积分到总积分
        total_score += heapq.heappop(queue)

    # 输出可以获得的最多积分
    print(total_score)

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

C语言

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

// 定义任务结构体
typedef struct {
    int deadline; // 任务的截止时间
    int score;    // 完成任务后获得的分数
} Task;

// 用于qsort的比较函数，按照任务的截止时间升序排序
int compare(const void *a, const void *b) {
    return ((Task *)a)->deadline - ((Task *)b)->deadline;
}
```

```
14
15  // 堆调整函数，用于维护小顶堆的性质
16  void heapify(Task *heap, int size) {
17      for (int i = size / 2 - 1; i >= 0; i--) {
18          int smallest = i;           // 假设当前节点是最小的
19          int left = 2 * i + 1;       // 左子节点索引
20          int right = 2 * i + 2;      // 右子节点索引
21
22          // 如果左子节点存在且小于当前最小节点，则更新最小节点
23          if (left < size && heap[left].score < heap[smallest].score) {
24              smallest = left;
25          }
26          // 如果右子节点存在且小于当前最小节点，则更新最小节点
27          if (right < size && heap[right].score < heap[smallest].score) {
28              smallest = right;
29          }
30
31          // 如果最小节点不是当前节点，交换它们，并递归调整被交换的子树
32          if (smallest != i) {
33              Task temp = heap[i];
34              heap[i] = heap[smallest];
35              heap[smallest] = temp;
36
37              heapify(heap, smallest);
38          }
39      }
40  }
41
42  int main() {
43      int n, T;
44      scanf("%d", &n); // 读取任务数量
45      scanf("%d", &T); // 读取时间限制
46
47      Task tasks[n];
48      // 读取每个任务的截止时间和分数
49      for (int i = 0; i < n; i++) {
50          scanf("%d %d", &tasks[i].deadline, &tasks[i].score);
51      }
52
53      // 对任务按截止时间进行排序
54
```

```
    qsort(tasks, n, sizeof(Task), compare);

    Task heap[T]; // 创建一个大小为T的小顶堆
    int size = 0; // 初始化堆的大小
    for (int i = 0; i < n; i++) {
        // 如果堆的大小小于当前任务的截止时间，则将任务加入堆中
        if (size < tasks[i].deadline) {
            heap[size++] = tasks[i];
            heapify(heap, size);
        // 如果堆不为空且堆顶任务的分数小于当前任务的分数，则替换堆顶任务
        } else if (size > 0 && heap[0].score < tasks[i].score) {
            heap[0] = tasks[i];
            heapify(heap, size);
        }
        // 如果堆的大小超过了时间限制，则移除堆顶元素
        if (size > T) {
            heap[0] = heap[--size];
            heapify(heap, size);
        }
    }

    int totalScore = 0;
    // 计算堆中所有任务的分数总和
    for (int i = 0; i < size; i++) {
        totalScore += heap[i].score;
    }

    // 输出总分
    printf("%d\n", totalScore);

    return 0;
}
```

完整用例

用例1

4

3

1 2
1 3
1 4
1 5

用例2

4
3
1 2
2 3
3 4
4 5

用例3

4
3
1 5
2 4
3 3
4 2

用例4

3
3
1 5
2 10
3 15

用例5

2
3
1 5
2 10

用例6

5
3
1 5
2 10
3 15
4 20
5 25

用例7

4
3
1 10
2 10
3 10
4 10

用例8

4
3
1 0
2 0
3 0
4 0

用例9

1
1
1 0

用例10

1
1
1 100

文章目录


机考真题
华为OD
No No No
CSDN @算法大师