24、高效的任务规划，考点 or 实现——动态规划+贪心思维

题目描述

- 你有 n 台机器，编号为 1~n，每台都需要完成一项工作，机器经过配置后都能完成独立完成一项工作。
- 假设第 i 台机器你需要花 B 分钟进行设置，然后开始运行，J 分钟后完成任务。
- 现在，你需要选择布置工作的顺序，使得用最短的时间完成所有工作。
- 注意，不能同时对两台进行配置，但配置完成的机器们可以同时执行他们各自的工作。

输入描述

- 第一行输入代表总共有 M 组任务数据（1<M<=10）。
- 每组数第一行为一个整数指定机器的数量 N（0<N<=1000）。
- 随后的 N 行每行两个整数，第一个表示 B（0<=B<=10000），第二个表示 J（0<=J<=10000）。
- 每组数据连续输入，不会用空行分隔。各组任务单独计时。

输出描述

- 对于每组任务，输出最短完成时间，且每组的结果独占一行。例如，两组任务就应该有两行输出。

用例

输入	1 1 2 2
输出	4
说明	第一行1：为一组任务， 第二行1：代表只有一台机器， 第三行2 2：表示该机器配置需2分钟，执行需2分钟。

输入	2 2 1 1 2 2 3 1 1 2 2 3 3
输出	4 7
说明	第一行2：代表两组任务， 第二行2：代表第一组任务有2个机器， 第三行1 1：代表机器1配置需要1分，运行需要1分， 第四行2 2：代表机器2配置需要2分，运行需要2分， 第五行3：代表第二组任务需要3个机器， 第6-8行分别表示3个机器的配置与运行时间。

题目解析

对于JavaScript Node模式来说，本题有两个难点，第一个难点是输入的获取。

如果基于 rl.on('line', line => {}) 事件监听的方式，来获取输入的话，每次事件回调执行完，我们得到的输入数据都将会丢失，因此我们需要将其缓存，但是一旦输入数据被缓存，它和后面的数据就没有了连贯性，比如我们获取到第一行m后，将其缓存在全局变量中，但是下一次获取新的输入时，我们如何知道新输入是什么呢?

在简单输入获取逻辑中，我们可以依赖于输入行数来判断，但是本题在获取完m后，并不能知道还会有多少行输入，因为m只是任务数，而具体多少行输入，还取决于每个任务有几台机器。

因此，我们需要一种类似于，Java语言的Scanner的同步输入获取。

而JavaScript中想将异步操作（事件回调的执行可以看成是异步的）变为同步化执行，那就只能将异步操作封装进Promise，然后利用async，await来阻塞同步代码，达到异步操作同步化执行。

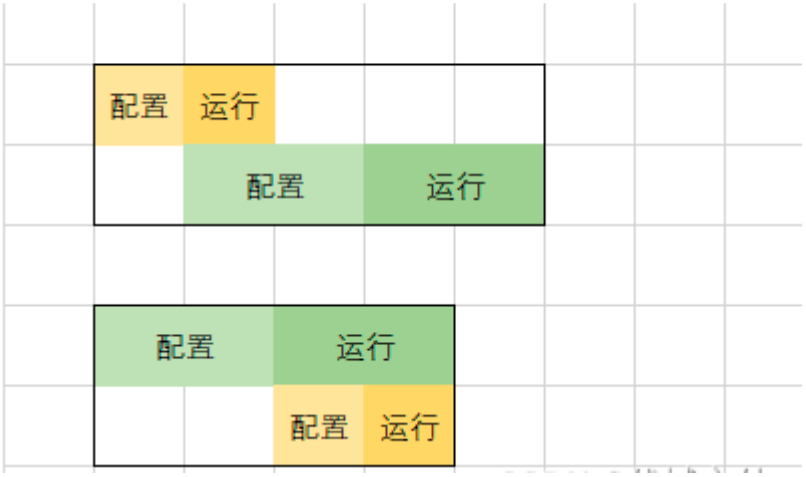
下面代码就是将line事件监听封装进promise对象，只有当有控制台输入时，才会将promise对象的状态更新为fulfilled

```
function getLine() {
  return new Promise((resolve) => {
    rl.on("line", (line) => {
      resolve(line);
    });
  });
}
```

因此，我们await 只会在promise对象变为fulfilled时，才会放行程序。

以上就是解决输入获取的方案说明。

下面说明本题算法逻辑：



如上图是两个机器执行的两种方案，我们可以发现

绿色机器先执行的话，总用时最少。

因为，绿色机器的运行时间更长，而橙色机器可以在绿色机器运行过程中完成配置和执行。

因此，我们很容易得出结论：如果想让任务总用时最少，则优先执行运行时间长的机器。这其实就是贪心思维。

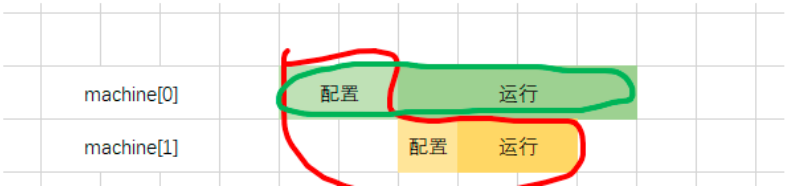
之后，就是计算多个机器工作时的最短时间了，

我们可以定义一个数组dp，dp[i]表示0~i台机器完成工作所需的最短时间。

我们假设machine[i] = [config, run]，即第i台机器需要配置时间config，运行时间run。

因此 dp[0] = machine[0][0] + machine[0][1]

而 dp[i] = Math.max(dp[i-1], dp[i-1] - machine[i-1][1] + machine[i][0] + machine[i][1])



如图所示，dp[1] = Math.max(dp[0], dp[0] - machine[0][1] + machine[1][0] + machine[1][1])

绿色标记和图中绿色圈对应，红色标记和图中红色圈对应。

JavaScript算法源码

```javascript
/* JavaScript Node ACM模式 控制台输入获取 */
const readline = require("readline");

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

// 将readline获取控制台输入 这种异步操作promisify，后面通过async await，即可实现同步获取控制台输入
function getLine() {
  return new Promise((resolve) => {
    rl.on("line", (line) => {
      resolve(line);
    });
  });
}

async function init() {
  // 同步控制台输入获取
  const m = (await getLine()) - 0; // 获取总任务数m
  const tasks = [];

  // 循环m次,每次获取一个任务下的机器详情
  for (let i = 0; i < m; i++) {
    const n = (await getLine()) - 0; //获取机器数n
    const machines = [];

    //循环n次，每次获取一个机器的 配置时间、运行时间
```

```
    //循环n次，每次获取一个机器的 配置时间、运行时间
    for (let j = 0; j < n; j++) {
      const machine = await getLine(); // 获取机器的 配置时间、运行时间
      machines.push(machine.split(" ").map(Number));
    }

    // tasks中存放的是task，每个task下又存放多个机器信息，每个机器信息包括：配置时间、运行时间
    tasks.push(machines);
  }

  // 业务逻辑
  for (let task of tasks) {
    // 将每个任务中的机器工作顺序，按照运行时间降序排序
    task.sort((a, b) => b[1] - a[1]);

    const dp = [];
    // dp[i]表示第i个机器完成工作的最少用时
    dp[0] = task[0][0] + task[0][1];

    // 下面这段逻辑情况题解图示
    for (let i = 1; i < task.length; i++) {
      dp[i] = Math.max(
        dp[i - 1],
        dp[i - 1] - task[i - 1][1] + task[i][0] + task[i][1]
      );
    }

    console.log(dp.at(-1));
  }
}
```

```
}

init();
```

Java算法源码

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

public class Main {
  // 输入获取
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    int m = sc.nextInt();

    int[][][] tasks = new int[m][][];

    for (int i = 0; i < m; i++) {
      int n = sc.nextInt();
      int[][] task = new int[n][2];
      for (int j = 0; j < n; j++) {
        task[j][0] = sc.nextInt();
        task[j][1] = sc.nextInt();
      }
      tasks[i] = task;
    }

    getResult(tasks);
  }

  // 算法入口
  public static void getResult(int[][][] tasks) {
    for (int[][] task : tasks) {
      // 将每个任务中的机器工作顺序，按照运行时间降序排序
      Arrays.sort(task, (a, b) -> b[1] - a[1]);

      int n = task.length;

      // dp[i]表示第i个机器完成工作的最少用时
      int[] dp = new int[n];
      dp[0] = task[0][0] + task[0][1];

      // 下面这段逻辑情况题解图示
      for (int i = 1; i < n; i++) {
        dp[i] = Math.max(dp[i - 1], dp[i - 1] - task[i - 1][1] + task[i][0] + task[i][1]);
      }

      System.out.println(dp[n - 1]);
    }
  }
}
```

Python算法源码

```python
# 输入获取
m = int(input())

tasks = [[] for _ in range(m)]

for i in range(m):
    n = int(input())
    task = [[] for _ in range(n)]
    for j in range(n):
        task[j] = list(map(int, input().split()))
    tasks[i] = task


# 算法入口
def getResult():
    for task in tasks:
        # 将每个任务中的机器工作顺序，按照运行时间降序排序
        task.sort(key=lambda x: -x[1])

        n = len(task)

        # dp[i]表示第i个机器完成工作的最少用时
        dp = [0] * n
        dp[0] = task[0][0] + task[0][1]

        # 下面这段逻辑情况题解图示
        for i in range(1, n):
            dp[i] = max(dp[i - 1], dp[i - 1] - task[i - 1][1] + task[i][0] + task[i][1])

        print(dp[n - 1])


# 算法调用
getResult()
```