题目描述：
让我们来模拟一个工作队列的运作，有一个任务提交者和若干任务执行者，执行者从 1 开始编号。

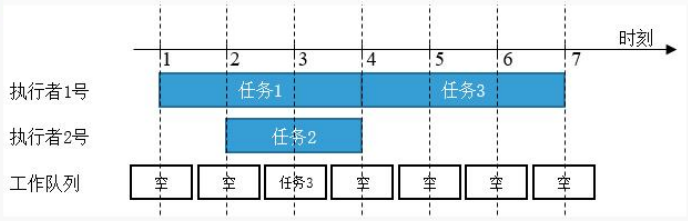
提交者会在给定的时刻向工作队列提交任务，任务有执行所需的时间，执行者取出任务的时刻加上执行时间即为任务完成的时刻。
执行者完成任务变为空闲的时刻会从工作队列中取最老的任务执行，若这一时刻有多个空闲的执行者，其中优先级最高的会执行这个任务。编号小的执行者优先级高。初始状态下所有执行者都空闲。
工作队列有最大长度限制，当工作队列满而有新的任务需要加入队列时，队列中最老的任务会被丢弃。
特别的，在工作队列满的情况下，当执行者变为空闲的时刻和新的任务提交的时刻相同时，队列中最老的任务被取出执行，新的任务加入队列。

输入描述：
输入为两行。第一行为 2N 个正整数，代表提交者提交的 N 个任务的时刻和执行时间。第一个数字是第一个任务的提交时刻，第二个数字是第一个任务的执行时间，以此类推。用例保证提交时刻不会重复，任务按提交时刻升序排列。

第二行为两个数字，分别为工作队列的最大长度和执行者的数量。

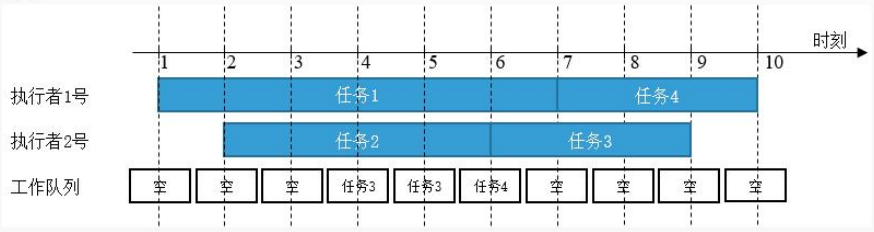
两行的数字都由空格分隔。N 不超过 20，数字为不超过 1000 的正整数。
输出描述：
输出两个数字，分别为最后一个任务执行完成的时刻和被丢弃的任务的数量，数字由空格分隔。
补充说明：

示例1
输入：1 3 2 2 3 3
3 2
输出：7 0
说明：



有两个执行者，执行者1号在时刻1取出了第1个任务，该任务需时为3，故1号会在时刻4完成任务；执行者2号在时刻2取出了第2个任务，该任务需时为2，故2号也会在时刻4完成任务；第3个任务在时刻3提交，但此时没有空闲的执行者，故缓存在队列中，直到时刻4两个执行者都变为空闲，此时执行者1号会取出这个任务，该任务需时为3，故会在时刻7完成任务。期间没有任务被丢弃。

示例2
输入：1 6 2 4 4 3 6 3
1 2
输出：10 0
说明：



有两个执行者，执行者1号在时刻1取出了第1个任务，该任务需时为6，故1号会在时刻7完成任务；执行者2号在时刻2取出了第2个任务，该任务需时为4，故2号会在时刻6完成任务；第3个任务在时刻4提交，但此时没有空闲的执行者，故缓存在队列中。时刻6执行者2号变为空闲，同时第4个任务被提交，此时队列已满。按特别说明，第4个任务加入队列，队列中最老的第3个任务被取出执行并在时刻9完成。最后执行者1号在时刻7取出第4个任务并在时刻10完成，期间没有任务被丢弃。

示例3
输入：1 6 2 4 3 3 4 3 6 3
1 2
输出：10 1
说明：

大致的流程和示例2类似，只是第4个任务在时刻4被提交时，没有空闲的执行者，而且工作队列已经缓存了第3个任务满了，故第3个任务被丢弃。

// 本题为考试单行多行输入输出规范示例，无需提交，不计分。

```
#include <iostream>
#include <deque>
#include <queue>
#include <functional>
using namespace std;
const int N = 100;
int start[N];
int take_time[N];
struct job{
	int start;
	int take_time;
};
std::deque<job> jobs_queue;
struct worker {
	int index, end_time;
};
bool operator<(worker a, worker b) {
	if (a.end_time == b.end_time) {
		return a.index > b.index;
	}
	return a.end_time > b.end_time;
}
```

```
std::priority_queue<worker> worker_table;
int main() {
	int top = 0;
	int a,b;
	while(cin >> a >> b) {
		start[top] = a;
		take_time[top] = b;
		top++;
	}
	int queue_size = start[top-1];
	int worker_number = take_time[top-1];
	for(int i=0;i<worker_number;i++) {
		worker_table.push({i,1});
	}
	top--;
	int loss = 0;
	int res_time = 0;

	for(int i=0;i<top;i++) {
//		std::cout << "push: " << start[i] <<" " << take_time[i] << "\n";
		jobs_queue.push_back({start[i], take_time[i]});

		while(jobs_queue.size() > 0) {
//			std::cout << "worker_table.top(): "<< worker_table.top().end_time << "\n";
			if(worker_table.top().end_time <= start[i]) {
				auto worker = worker_table.top();
				int end_time = max(worker.end_time, jobs_queue.front().start) + jobs_queue.front().take_time;
				worker_table.pop();
				worker_table.push({worker.index, end_time});
				res_time = max(res_time, end_time);
//				std::cout << "work: end: " << end_time << "\n";

				jobs_queue.pop_front();
			} else{
				break;
			}
		}

		while(jobs_queue.size() > queue_size) {
			loss++;
//				std::cout << "diu: " << jobs_queue.front().start <<" " << jobs_queue.front().take_time << "\n";
			jobs_queue.pop_front();
```

```
            }
        }
        while(jobs_queue.size() > 0) {
//                      std::cout << "worker_table.top(): "<< worker_table.top().end_time << "\n";
                    auto worker = worker_table.top();
                    int end_time = max(worker.end_time, jobs_queue.front().start) +
jobs_queue.front().take_time;
                    worker_table.pop();
                    worker_table.push({worker.index, end_time});
//                      std::cout << "work: end: " << end_time << "\n";
                    res_time = max(res_time, end_time);

                    jobs_queue.pop_front();
        }
        std::cout << res_time << " " << loss << std::endl;
}
```