Solution — Moving books

1 Modeling

We are given two lists of integers $s_0, \ldots, s_{n-1}$ and $w_0, \ldots, w_{m-1}$. Here s_i is the *strength* of the i-th person and w_i is the *weight* of the i-th box. Person i is able to move box j if and only if $s_i \geq w_j$.

The problem asks for the minimum integer t such that the given people can move all the given boxes within t minutes. Here it is important to know that a person needs two minutes to move a box (independent of his strength and the weight of the box), and needs another minute before he can move the next box. Multiple people can move multiple boxes in parallel, but a single person can only carry one box at any given time. It might be that it is not possible for the people to move all boxes (because some box is heavier then the strongest person is strong). In that case, we are to output the string *impossible*.

One observation that we can make right away is that in any optimal solution, the people will move the boxes in discrete *rounds*, where a round consists of a certain number of people moving boxes in parallel. From now on, instead of counting the time in minutes, we count the number of rounds, which seems more natural. Our goal is to compute the least integer r such that the people can move all boxes in r rounds. Then the number of minutes (i.e., the number we should output) is $3r - 1$, because in all rounds we need two minutes to move the boxes, and in all but the last round, we need an additional minute before the next round can start.

Constraints Before thinking more about the problem, it is a good idea to have a closer look at the constraints. For simplicity, let us denote by $N := n + m$ the size of the input and let $W := \max_i s_i \geq \max_i w_i$ be the maximum strength of a person. The problem description guarantees that $N \leq 1\,000\,000$ and that $W \leq 300\,000$. This suggests that in order to get 100 points, we should to find a solution that runs in time $O(N+W)$ or at worst $O(N \log N + W \log W)$. A quadratic time solution will certainly be too slow.

In the first testset, we are guaranteed that $N \leq 10\,000$ and that $W \leq 100$. This means that here we can probably get away with a solution that runs in time $O(N^2)$.

In the second testset, we are guaranteed that $W \leq 100$. This could mean that one can get these these points with a solution that runs in time $O(WN)$.

2 Algorithm design

First, note that the output is *impossible* if and only if $\max_i w_i > \max_i s_i$. Since this condition is easily checked for, we will assume for the rest of this section that $\max_i w_i \leq \max_i s_i$. In this case, the least number r of rounds in which all boxes can moved is an integer between 1 and m (because at least one box is moved every round).

A natural start is to try finding a reasonable strategy which allows the people to move the boxes in as few rounds as possible. It seems that the following greedy strategy is a good candidate: within a given round, we process the people one by one, making each person move the heaviest box that he is able to lift and that has not already been moved, provided such a box exists.

Just as in the example problems covered in the lecture, it can be proved by an exchange argument that this strategy is indeed optimal. Here, we omit this proof. In any case, it is intuitively clear that if a person switches to a heavier box, he only makes life easier for those that come after him.

The greedy strategy can be described by the following pseudocode:

```
B = {0,...,m-1}
while B ≠ ∅:
    for each 0 <= i < n:
        J = {j ∈ B | w[j] <= s[i]}
        if J ≠ ∅:
            j = any j in J such that w[j] is maximal
            B = B \ {j}
```

Note that the for loop simulates a single round of the greedy strategy. An obvious idea is that we simply run this strategy on our input and count how many rounds it takes (i.e., how often the while loop is repeated).

We can represent the set `B` by a vector of m zeroes and ones, each entry representing whether `w[i]` belongs to `B` or not. Removing an element from `B` is then also easy. The check whether `B` is empty can be done in constant time if we keep track of the size of `B` in a separate variable.

A straightforward implementation of the code inside the for loop will run in time $\Omega(N)$ in the worst case, since it requires looking at each weight `w[j]`. This would result in time $\Omega(N^2)$ to process a single round. Since in the worst case there could be $m = \Omega(N)$ rounds until all boxes are moved, this solution takes time $\Omega(N^3)$. This is too slow to get any points in this problem.

$O(N^2)$ solution (20 points) One reason why the naive solution is slow is that is needs quadratic time to simulate a single round. We would gain a lot if we could do the same work in time $O(N)$. There are several ways to achieve this. One way is to do a *precomputation step* in which the people are sorted from strongest to weakest and the boxes are sorted from heaviest to lightest. As mentioned in the lecture slides, sorting is a common precomputation step; this holds especially when designing greedy algorithms.

The sorting takes an additional time of $O(N \log N)$, but it will allow us to simulate a single round in time $O(N)$. The following pseudocode shows how a single round can be simulated, provided the strengths and weights are sorted.

```
while B ≠ ∅:
    i = j = 0
    while i < n and j < m:
        // Invariant: all boxes 1,...,j-1 are either already moved or they are too
        // heavy for person i.
        if j ∈ B and w[j] <= s[i]:
            B = B \ {j}
            i = i+1
        j = j+1
```

Because the strengths are sorted, it is easy to see that the invariant stated in the code is maintained throughout the execution. Moreover, because of this invariant and because the weights are sorted, one can see that this algorithm really implements the greedy strategy correctly, i.e., each person moves the heaviest remaining box that he is able to. The total running time is $O(N^2)$, which would give 20 points.

$O(NW)$ **solution (60 points)** For the first two test sets, we know that $W \leq 100$, which gave us hope that a $O(NW)$ solution could give 60 points. Indeed, there is another sort of precomputation which allows us to simulate a single round in time $O(W)$, thus giving an algorithm that runs in time $O(NW)$.

The idea is the following. We precompute two numbers `p[w]` and `b[w]` for each weight $1 \leq w \leq W$, where `p[w]` is the number of people of strength exactly w and `b[w]` is the number of boxes of weight exactly w. This precomputing can easily be done in time $O(N)$. Assuming that we these values, it is possible to handle all box movements for boxes with the same weight in constant time (per round). The following pseudocode shows how to simulate a single round:

```
1 useful = 0
2 w = W
3 while w >= 0:
4     useful += p[w]
5     moved = min(b[w], useful)
6     b[w] -= moved
7     useful -= moved
8     w = w-1
```

After executing line 4 in the while loop, the variable `useful` contains the number of people that have strength at least w and that are not yet moving a box in this round. We then compute the number of boxes of weight w moved by these people as `min(b[w],useful)` and adjust the number `b[w]` of remaining boxes of that weight and `useful` accordingly.

It is not too hard to turn this into a full solution which gives 60 points.

$O(N \log N)$ **solution (100 points)** Let us write x_r for the number of boxes that are moved in round r by the greedy strategy. We want to find a method to simulate a given round r in time $O(x_r \log N)$. Then, since the sum of all x_r is $m = O(N)$, the total running time would be $O(N \log N)$.

However, looking at the pseudocode for the greedy strategy, it seems that for a single round, we need to iterate at least over all people. So how can we do better than $O(N)$? Again, sorting comes to the rescue. Assume that the strengths are sorted so that `s[0] >= ... >= s[n-1]`. In that case, we can stop processing people once we have found the first person that is too weak to carry any remaining box (since then all subsequent people will also be too weak). This is exemplified in the following pseudocode:

```
1 B = {0,...,m-1}
2 while B ≠ ∅:
3     for each 0 <= i < n:
4         J = {j ∈ B | w[j] <= s[i]}
5         if J ≠ ∅:
6             j = any j in J such that w[j] is maximal
7             B = B \ {j}
8         else:
9             break
```

It is easy to see that the inner for loop is repeated only $x_r + 1$ times. It remains to find an efficient solution for handling the code inside the this loop. For this, we need a more efficient data structure for storing the box weights. In particular, we want to be able to carry out the following two operations efficiently: (1) removing a weight, and (2) finding the heaviest weight which is still at most $s[i]$.

Fortunately for us, there is a data structure in the C++ STL that can do both of these things efficiently: `std::multiset`. Both removing elements and answering queries of the form in (2)

takes $O(\log N)$ time in a `std::multiset` with N elements. Using this data structure in the algorithm outlined above results in a $O(N \log N)$ solution that gives 100 points.

Binary search solution (100 points) It is also possible to solve this problem using the binary search approach explained in the lecture slides: we first solve the simpler problem of determining whether the given people can move all boxes within at most r rounds, for a fixed r. Then we perform a binary search to find the least r for which this is the case.

In fact, let us consider before that the even simpler problem of determining whether the people can move all boxes in a single round. This easy to do in time $O(N)$. In fact, from the $O(N^2)$ solution above we already know how to simulate a single round in time $O(N)$ if the strengths and the weights are sorted. The problem of checking whether all boxes can be moved in a single round is very similar. The pseudocode for this looks slightly simpler because it now unnecessary to track the set `B`. It is assumed that the weights and the strengths are sorted in decreasing order.

```
def possible_in_1_round:
    i = j = 0
    while i < n and j < m:
        if w[j] <= s[i]:
            i = i+1
            j = j+1
        else:
            return false
    return (j >= m)
```

If we want to check whether it is possible to move all boxes in at most r rounds, we can conceptually replace each person by r people with the same strength, and check whether these people could move all boxes in a single round. Of course, we do not want to actually create a vector of rn strengths, since then the running time would be in the order of rN (and we want $O(N)$). The following solution achieves the same thing in linear time:

```
def possible_in_r_rounds:
    i = j = 0
    while i < n and j < m:
        if w[j] <= s[i]:
            i = i+1
            j = j+r
        else:
            return false
    return (j >= m)
```

Note that the `possible_in_r` predicate still runs in time $O(N)$. Having this fact in mind, we try to guess the optimal number of rounds with binary search. Since the final number of rounds is at most $m = O(N)$, we will obtain a solution that runs in time $O(N \log N)$.

3 Implementation

We only give implementation details for the solution using `std::multiset`. We will store all box weights in a container `ws` of type `std::multiset<int, std::greater<int> >`. Note that we must choose `std::multiset` and not `std::set` because the same weight might appear multiple times. The `std::greater<int>` means that the natural order on the weights is assumed to be reversed. This means that when we call `ws.lower_bound(s)`, we obtain an iterator to a weight that is as large as possible subject to being at most `s` (without the `std::greater<int>` we would

get a weight that is as small as possible while being at least s). This function runs in time $O(\log N)$.

For this solution, we also need to sort the strengths in decreasing order. This can be achieved as follows:

```
std::sort(s.begin(), s.end(), std::greater<int>());
```

Again, the `std::greater<int>()` specifies that the order should be decreasing, not increasing. Translating the pseudocode given above into C++ is straightforward:

```
B = {0,...,m-1}
while B ≠ ∅:
    for each 0 <= i < n:
        J = {j ∈ B | w[j] <= s[i]}
        if J ≠ ∅:
            j = any j in J such that w[j] is maximal
            B = B \ {j}
        else:
            break
```

becomes

```
std::multiset<int, std::greater<int> > ws;
for (int i = 0; i < m; ++i) ws.insert(w[i]);
while (!ws.empty()) {
    for (int i = 0; i < n; ++i) {
        auto j = ws.lower_bound(s[i]);
        if (j != ws.end()) {
            ws.erase(j);
        } else {
            break;
        }
    }
}
```

Note the fact that `ws` contains weights, while `B` contains indices. Moreover, the two steps of first finding `J` and then `j` are handled in a single step by the call to `lower_bound`.

4 Appendix

The following code is an implementation of $O(N \log N)$ solution which uses `std::multiset` data structure as explained in the previous section.

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

int main()
{
  int t;
  std::ios_base::sync_with_stdio(false);
  std::cin >> t;

  while (t--) {
    int n, m;
    std::cin >> n >> m;
    std::vector<int> s(n), w(m);
    std::multiset<int, std::greater<int> > ws;

```

```
    for(int i = 0; i < n; ++i) std::cin >> s[i];
    for(int i = 0; i < m; ++i) std::cin >> w[i];

    // First check whether the answer is impossible.
    int maxw = -1;
    int maxs = -1;
    for (int i = 0; i < n; ++i)
      maxs = std::max(maxs, s[i]);
    for (int i = 0; i < m; ++i)
      maxw = std::max(maxw, w[i]);
    if (maxw > maxs) {
      std::cout << "impossible" << std::endl;
      continue; // Go to next test case.
    }

    // Precomputation: sort the strengths.
    std::sort(s.begin(), s.end(), std::greater<int>());

    // Now simulate the greedy algorithm.
    for (int i = 0; i < m; ++i) ws.insert(w[i]);
    int r = 0;
    while (!ws.empty()) {
      ++r;
      for (int i = 0; i < n; ++i) {
        auto b = ws.lower_bound(s[i]);
        if (b != ws.end()) {
          ws.erase(b);
        } else {
          break;
        }
      }
    }

    // Output the correct answer.
    std::cout << 3*r -1 << std::endl;
  }

  return 0;
}
```