You will implement an “order-filling algorithm” and unit tests for it. Other group members will independently develop a “unique-identifier data structure” and a “warehouse model.” The sample subset.c file is about 120 lines (this does not include other files), but you are given about 50 of those lines.

Requirements:

- Put your code in two files, subset.c and subset_test.c. Both should include subset.h. Write an appropriate Makefile.
- subset.h (provided) should have just these declarations plus typical header-file stuff:

```c
struct SubsetData {
    int num_orders; // length of orders
    int num_parts; // length of orders[i] and inventory
    int ** orders;
    int * inventory;
};
typedef struct SubsetData * sdata_t;

struct SubsetAns {
    int count; // length of orders
    int * orders;
    int * inventory;
};

struct SubsetAns find_orders(sdata_t d);
```

- The purpose of find_orders is to find a large subset of the “orders” that can be “fulfilled” given the “inventory”. The orders are in the array d->orders and each order is an array of of length num_parts where the jth entry is the quantity of part j necessary for the order. That is, d->orders[i][j] is the quantity of part j used in order i.
  The ith entry of d->inventory is the quantity of part i available. The selected subset of orders cannot use more of a part than is in the inventory. You must find two different subsets and choose the better one as explained below.

- In the result (struct SubsetAns), the first count elements of orders are the order numbers (indices of the orders field in struct SubsetData) that are in the subset of fulfilled orders. The inventory is the inventory after fulfilling these orders. Do not mutate any of the data pointed to by the argument of find_orders.

- In a helper function, implement a greedy algorithm to find a subset as follows:
  - Let the result inventory start as a copy of the initial inventory.
  - For orders 0, 1, 2, ... in order (up to the total number of orders), if the order can be fulfilled, add it to the subset and update the result inventory appropriately.

Note the code provided to you is not helpful for this algorithm. Also note that if an order is not in the subset, it must not affect the result inventory. This algorithm is fast (it considers each order only once), but may choose a much smaller subset than is possible.

- In another helper function, implement a maximal algorithm as follows:
For each subset-size 1, 2, ... in order (up to the total number of orders), try to find a subset of orders of this size that can be fulfilled, but stop once a size fails (since no larger size can succeed).

If a subset of size \( n \) is not found, stop and return a subset of size \( n - 1 \). (This will require keeping a copy of the orders and result-inventory for that smaller subset; be careful to keep that memory separate but do not leak space.)

Else remember a subset and result-inventory for size \( n \) and continue with size \( n + 1 \).

Because this algorithm could take a very long time (the number of subsets is exponential in the number of orders), the algorithm for finding a subset of size \( n \) should simply fail if more than a couple seconds have elapsed since the overall maximal algorithm began.

Note the code for finding a subset of size \( n \), including the code for keeping track of elapsed time, has been written for you.

- In `find_orders`, call helper functions to do both the greedy and the maximal algorithm. Return the `struct SubsetAns` with the higher count (either is fine if the counts are equal). Be sure not to leak space (since one result’s subset and result-inventory will not be returned).

**Advice/Hints:**

- Understand the struct definitions and algorithms before you start coding.

- To develop a test where the maximal algorithm times out, you should need only a few dozen orders and an understanding of how the algorithm works.

- Do not worry that the maximal algorithm returns an array where higher-order numbers appear first; this does not matter.

- While the maximal algorithm can be done with four heap-allocated arrays (two for orders and two for result-inventories), it may be easier just to malloc/free on each iteration.

- For the result orders, it may be easiest to allocate an array of size `num_orders` even though the `count` may be smaller. This is fine (the amount of wasted space is small and would be reclaimed if the caller freed the array).

**Assessment and turn-in:**

Your solutions should be:

- Correct C code that compiles without warnings using `gcc -Wall` and does not have space leaks

- In good style, including indentation and line breaks

- Of reasonable size

Your test code should provide good **coverage**.

Use `turnin` for course `cse303` and project `hw5`. If you use late-days, use project `hw5late1` (for 1 late day) or `hw5late2` (for 2).