Sample output

```
n = 25
p = 5

Enter customer times:

 13 11 25 23 14 22  3 24 22  8 27  2 21 21  5 19  7 12 25  5 12 30  8 27  2

+++ Schedule 1
Counter  0:   0 (13)   6 ( 3)   8 (22)  14 ( 5)  16 ( 7)  17 (12)  23 (27)
Counter  1:   1 (11)   5 (22)  12 (21)  19 ( 5)  22 ( 8)
Counter  2:   2 (25)  10 (27)  18 (25)
Counter  3:   3 (23)   9 ( 8)  11 ( 2)  13 (21)  20 (12)  24 ( 2)
Counter  4:   4 (14)   7 (24)  15 (19)  21 (30)
+++ Bank finishes at time 89
+++ Total waiting time = 772

+++ Schedule 2
Counter  0:  21 (30)  12 (21)  17 (12)   1 (11)  19 ( 5)
Counter  1:  10 (27)   5 (22)  15 (19)  22 ( 8)  11 ( 2)
Counter  2:  23 (27)   8 (22)   4 (14)   9 ( 8)  14 ( 5)  24 ( 2)
Counter  3:   2 (25)   7 (24)   0 (13)  20 (12)   6 ( 3)
Counter  4:  18 (25)   3 (23)  13 (21)  16 ( 7)
+++ Bank finishes at time 79
+++ Total waiting time = 1076

+++ Schedule 3
Counter  0:  ...
Counter  1:  ...
Counter  2:  ...
Counter  3:  ...
Counter  4:  ...
+++ Bank finishes at time 79
+++ Total waiting time = 478
```

The State Bank of Greedland (SBG) is supposed to serve n customers on a day. All the customers come to the bank at the opening time ($t = 0$), and request services. Serving the request of the i-th customer takes time t_i (an integer, like number of minutes), and the bank knows these times from the very beginning. The bank has p processing counters. The objective of SBG is to schedule the customer requests to the counters in such a way that the bank finishes serving all the requests as early as possible.

Computing the best possible solution to this problem is believed to be very difficult. Several greedy strategies are known, that produce *good* solutions, that is, solutions within constant factors of the best possible solution. In this assignment, you implement some such strategies.

Part 1: In this part, you write a function *schedule1*() that, given n, $t_0, t_1, t_2, \ldots, t_{n-1}$, and p, produces a schedule for the counters following a greedy approach outlined now. Suppose that the first i requests $t_0, t_1, t_2, \ldots, t_{i-1}$ are already scheduled. That means that the p counters have some assigned loads, and finish the current loads at times $f_0, f_1, f_2, \ldots, f_{p-1}$, respectively. The request t_i is scheduled to the j-th counter if $f_j = \min(f_0, f_1, \ldots, f_{p-1})$. In case multiple counters finish at the same minimum time, take j as small as possible. The function *schedule1*() would return p arrays, each storing the computed schedule for a counter.

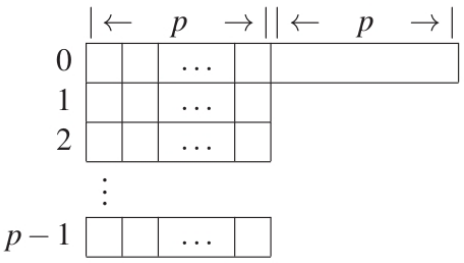
Use a two-dimensional array of integers to store a schedule. The i-th row starts with the number of requests scheduled to the i-th counter, and is followed by the indices of the requests scheduled to this counter. You may use a static array of predefined maximum dimensions or a dynamically allocated array. Do not use lists defined in the STL. You should be fully responsible for structuring your data.

2	0	8		
4	1	4	7	9
1	2			
3	3	5	6	

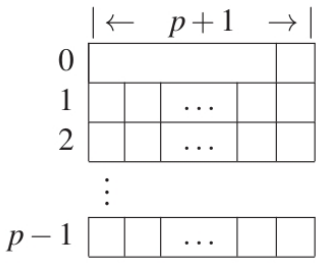
Alongside the scheduling function, write a function *printschedule*() that, given a schedule (a 2-D array), prints the allocation of the requests to the counters as given in the sample output below. An allocation is shown as a pair i (t_i). The index i is available from the schedule (the 2-D array), and t_i is read from the input array t. Write another function *finishingtime*() that, given a schedule, returns the finishing time of the bank.

Part 2: The last request t_i (say, served at Counter j) determines the finishing time of SBG. If t_i is a long request, whereas some other Counter k finishes earlier than Counter j after serving several short requests, swapping the short requests by the long request and redistributing these short requests among the available counters may improve the bank's finishing time. But that requires scheduling t_i *out of order*, that is, before some of $t_{i-1}, t_{i-2}, \ldots$. In other words, it is often possible to decrease the bank's finishing time by scheduling the longer requests earlier than the shorter ones.

To illustrate this concept, take $n = p^2 + 1$ with the first p^2 requests demanding time 1 each and the last request demanding time p. The schedule of Part 1 evenly distributes the first p^2 requests to the p counters, and assigns the last request to the first counter, so the bank finishes at time $2p$. The schedule of Part 2 would assign the long request and one short request to the first counter, and evenly distribute the remaining $p^2 - 1$ short requests to the remaining $p - 1$ counters. Now, all the counters finish simultaneously at time $p + 1$.



(a) Part 1



(b) Part 2

Write a function *schedule2*() that first sorts the times $t_0, t_1, t_2, \ldots, t_{n-1}$ in the decreasing (or non-increasing) order, and uses this sorted sequence to schedule following the same strategy as explained in Part 1. Write your own function for merge sorting an array of integers in the non-increasing order. Do not use a library function (like *qsort*() already implemented in the standard libraries of C and C++).

Part 3: Let the i-th customer be served by Counter j starting at time s_i. We call s_i the *waiting time* of Customer i. It is equal to the sum of the times taken by the requests served at Counter j before Customer i is served. The *total waiting time* is defined as

$$T = \sum_{i=0}^{n-1} s_i.$$

Larger values of T imply higher dissatisfaction of the customers. Write a function *totalwaittime*() that, given a schedule (in the form of a 2-D array), returns the value of T.

Write an $O(n)$-time function *schedule3*() to minimize customer dissatisfaction (the value of T) without increasing the bank's finishing time as achieved by the schedule of Part 2.

The *main*() function

- Read n (the number of customers) and p (the number of counters) from the user.
- Store in an array $t_0, t_1, t_2, \ldots, t_{n-1}$ (positive integers) to be supplied by the user.
- Call *schedule1*(), and print the returned schedule using *printschedule*(), the bank's finishing time by calling *finishingtime*(), and the total waiting time of the customers by calling *totalwaittime*().
- Repeat for the schedule produced by *schedule2*().
- Repeat for the schedule produced by *schedule3*().