Question 2b: what if we ask you to return a **random largest number's index**

For example, given a **stream** like "1, 2, 5a, 3, 4, 3, 4, 5b", you may return 5a or 5b's index randomly.

	1	2	5a	3	4	3	4	5b
index	0	1	2	3	4	5	6	7
{2,7}								i

cur_max: 5
cur_max_count: 2
cur_max_sample_index: {2,7}

array[i] < cur_max -- continue
array[i] > cur_max -- cur_max = array[i], count = 1, cur_max_sample_index = i
array[i] == cur_max -- count++, do reservior sampling in this case.

Question 3a: How to design a random number generator Random(7), with **Random**(5).

Random(5) -- ⅕ return 0, 1, 2, 3, 4
Random(7) -- 1/7 return 0, 1, 2, 3, 4, 5, 6 - uniformly distributed

`Reversed direction (is easy):`

`Random(7) → Random(5)`

`0   1   2   3  4   5  6`

1/7 1/7 .. 1/7

```
int ret = Random(7);
while (ret >= 5) {
        ret = Random(7);
}
return ret;
```

do random(7) until the result is in the range of [0,1,2,3,4], return the result.

0,1,5,6,3,2,4,6,1,...

Probability **Normalization**

P(0 can be returned) = 1/7 / **5/7** = 1/5

P(1 can be returned) = 1/7 / 5/7 = 1/5

P(2 can be returned) = 1/7 / 5/7 = 1/5

P(3 can be returned) = 1/7 / 5/7 = 1/5

P(4 can be returned) = 1/7 / 5/7 = 1/5

`How to Random(5) → Random(7)`

How to use Random(5) to get a Random(x), x > 7.

step 1.

Random(25) - how to implement Random(25) using Random(5)

```
5 * Random(5) + Random(5) = Random(25)
      |                   |
  random row        random column
```

For each number generated with the method above, its probability = 1/25

index	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	**12**	13	14
3	15	16	17	18	19
4	20	21	22	23	24

```
5 * row + col
```

Or, you can think this way: 0 - 24 are all the 2 digits radix base 5 numbers: [0-4][0-4]
Random(5) for the first digit
Random(5) for the second digit

===> k digits base 5 numbers:[0-4][0-4]....[0-4]
Random(5) call k times, we can get **Random(5^k).**

Question 4 : Given a **data flow,** how to keep track of the **median** of the numbers read so far? space: **O(n)**.

median - after sorting the sequence, the element at the middle position.
{1, 2, 3, **4,** 5, 6, 7} - median=4
{1, 2, 3, **4, 5**, 6, 7, 8} - median = (4+5)/2 = 4.5

	5,	1,	2,	7,	4,	10,
median:	5,	3,	2,	3.5,	4,	4.5,

Follow up: Q4b

Delong (what if the number of element is tooooo large)

```
small  50% elements                 ||              large 50% elements

          Max_heap                                         Min_heap

xxxxxxxxxxxxxxxxxxxxxxX  xxxxxX      Yyyyyyyy Yyyyyyyyyyyyyyyyyyyyyyyy
                      60      99     190    200 180    200      300
```

1G memory

Follow up: Q4b

Delong (what if the number of element is tooooo large)

```
small  50% elements                 ||            large 50% elements

         Max_heap                                          Min_heap

xxxxxxxxxxxxxxxxxxxxxxX  xxxxxX     Yyyyyyyy Yyyyyyyyyyyyyyyyyyyyyyyy
                  59    60     99    100    170 180    200       300
```

1G memory

500M 250M + 250M buffer

[250m buffer] [60-99] [100-170] [250M buffer]

read 500M < 59 elements.

[500M < **59**] [500M, 60-170]

X

large data --> can not fit into memory --> part of the data should be on the disk.

==> I/O , comparing to memory, very expensive. --> best effort to retain the operations in memory.

==> single disk seek operation. ==> avoid such operation

==> batch process has better efficiency

1. once the memory is full, then we dump the buffer into the disk.
2. when the maximum value of the smaller half in memory < the maximum value on disk.