**Shuffle a given array**

Given an array, write a program to generate a random permutation of array elements. This question is also asked as “shuffle a deck of cards” or “randomize a given array”.

[Fisher–Yates shuffle Algorithm](http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle#The_modern_algorithm) works in O(n) time complexity. The assumption here is, we are given a function rand() that generates random number in O(1) time.  
The idea is to start from the last element, swap it with a randomly selected element from the whole array (including last). Now consider the array from 0 to n-2 (size reduced by 1), and repeat the process till we hit the first element.

Following is the detailed algorithm

To shuffle an array a of n elements (indices 0..n-1):

for i from n - 1 downto 1 do

j = random integer with 0 <= j <= i

exchange a[j] and a[i]

**Reservoir Sampling**

[Reservoir sampling](http://en.wikipedia.org/wiki/Reservoir_sampling) is a family of randomized algorithms for randomly choosing *k* samples from a list of *n* items, where *n* is either a very large or unknown number. Typically *n* is large enough that the list doesn’t fit into main memory. For example, a list of search queries in Google and Facebook.

So we are given a big array (or stream) of numbers (to simplify), and we need to write an efficient function to randomly select *k* numbers where *1 <= k <= n*. Let the input array be *stream[].*

It **can be solved in O(n) time**. The solution also suits well for input in the form of stream. The idea is similar to [this](http://www.geeksforgeeks.org/archives/25111) post. Following are the steps.

**1)** Create an array reservoir[0..k-1] and copy first k items of stream[] to it.  
**2)** Now one by one consider all items from (k+1)th item to nth item.  
…**a)** Generate a random number from 0 to i where i is index of current item in stream[]. Let the generated random number is j.  
…**b)** If j is in range 0 to k-1, replace reservoir[j] with arr[i]

Code :

// A function to randomly select k items from stream[0..n-1].

void selectKItems(int stream[], int n, int k)

{

    int i;  // index for elements in stream[]

    // reservoir[] is the output array. Initialize it with

    // first k elements from stream[]

    int reservoir[k];

    for (i = 0; i < k; i++)

        reservoir[i] = stream[i];

    // Use a different seed value so that we don't get

    // same result each time we run this program

    srand(time(NULL));

    // Iterate from the (k+1)th element to nth element

    for (; i < n; i++)

    {

        // Pick a random index from 0 to i.

        int j = rand() % (i+1);

        // If the randomly  picked index is smaller than k, then replace

        // the element present at the index with new element from stream

        if (j < k)

          reservoir[j] = stream[i];

    }

    printf("Following are k randomly selected items \n");

    printArray(reservoir, k);

}

Time Complexity: O(n)

**How does this work?**  
To prove that this solution works perfectly, we must prove that the probability that any item stream[i] where 0 <= i < n will be in final reservoir[] is k/n. Let us divide the proof in two cases as first k items are treated differently.

**Case 1: For last n-k stream items, i.e., for stream[i] where k <= i < n**   
For every such stream item stream[i], we pick a random index from 0 to i and if the picked index is one of the first k indexes, we replace the element at picked index with stream[i]

To simplify the proof, let us first consider the last item. The probability that the last item is in final reservoir = The probability that one of the first k indexes is picked for last item = k/n (the probability of picking one of the k items from a list of size n)

Let us now consider the second last item. The probability that the second last item is in final reservoir[] = [Probability that one of the first k indexes is picked in iteration for stream[n-2]] X [Probability that the index picked in iteration for stream[n-1] is not same as index picked for stream[n-2] ] = [k/(n-1)]\*[(n-1)/n] = k/n.

Similarly, we can consider other items for all stream items from stream[n-1] to stream[k] and generalize the proof.

**Case 2: For first k stream items, i.e., for stream[i] where 0 <= i < k**   
The first k items are initially copied to reservoir[] and may be removed later in iterations for stream[k] to stream[n].  
The probability that an item from stream[0..k-1] is in final array = Probability that the item is not picked when items stream[k], stream[k+1], …. stream[n-1] are considered = [k/(k+1)] x [(k+1)/(k+2)] x [(k+2)/(k+3)] x … x [(n-1)/n] = k/n

## Average of a stream of numbers

Given a stream of numbers, print average (or mean) of the stream at every point. For example, let us consider the stream as 10, 20, 30, 40, 50, 60, …

Average of 1 numbers is 10.00

Average of 2 numbers is 15.00

Average of 3 numbers is 20.00

Average of 4 numbers is 25.00

Average of 5 numbers is 30.00

Average of 6 numbers is 35.00

..................

To print mean of a stream, we need to find out how to find average when a new number is being added to the stream.

Solution :

Let n be the count, prev\_avg be the previous average and x be the new number being added. The average after including x number can be written as (prev\_avg\*n + x)/(n+1).

## Find the largest multiple of 3

Given an array of non-negative integers. Find the largest multiple of 3 that can be formed from array elements.

For example, if the input array is {8, 1, 9}, the output should be “9 8 1″, and if the input array is {8, 1, 7, 6, 0}, output should be “8 7 6 0″.

This problem can be solved efficiently with the help of O(n) extra space. This method is based on the following facts about numbers which are multiple of 3.

**1)** A number is multiple of 3 if and only if the sum of digits of number is multiple of 3. For example, let us consider 8760, it is a multiple of 3 because sum of digits is 8 + 7+ 6+ 0 = 21, which is a multiple of 3.

**2)** If a number is multiple of 3, then all permutations of it are also multiple of 3. For example, since 6078 is a multiple of 3, the numbers 8760, 7608, 7068, ….. are also multiples of 3.

**3)** We get the same remainder when we divide the number and sum of digits of the number. For example, if divide number 151 and sum of it digits 7, by 3, we get the same remainder 1.

What is the idea behind above facts?  
The value of 10%3 and 100%3 is 1. The same is true for all the higher powers of 10, because 3 divides 9, 99, 999, … etc.  
Let us consider a 3 digit number n to prove above facts. Let the first, second and third digits of n be ‘a’, ‘b’ and ‘c’ respectively. n can be written as

n = 100.a + 10.b + c

Since (10^x)%3 is 1 for any x, the above expression gives the same remainder as following expression

1.a + 1.b + c

So the remainder obtained by sum of digits and ‘n’ is same.

Following is a solution based on the above observation.

**1.** Sort the array in non-decreasing order.

**2.** Take three queues. One for storing elements which on dividing by 3 gives remainder as 0.The second queue stores digits which on dividing by 3 gives remainder as 1. The third queue stores digits which on dividing by 3 gives remainder as 2. Call them as queue0, queue1 and queue2

**3.** Find the sum of all the digits.

**4.** Three cases arise:  
……**4.1** The sum of digits is divisible by 3. Dequeue all the digits from the three queues. Sort them in non-increasing order. Output the array.

……**4.2** The sum of digits produces remainder 1 when divided by 3.  
Remove one item from queue1. If queue1 is empty, remove two items from queue2. If queue2 contains less than two items, the number is not possible.

……**4.3** The sum of digits produces remainder 2 when divided by 3.  
Remove one item from queue2. If queue2 is empty, remove two items from queue1. If queue1 contains less than two items, the number is not possible.

**5.** Finally empty all the queues into an auxiliary array. Sort the auxiliary array in non-increasing order. Output the auxiliary array.

NICE PROBLEM...

**Find the largest multiple of 2, 3 and 5**

An array of size n is given. The array contains digits from 0 to 9. Generate the largest number using the digits in the array such that the number is divisible by 2, 3 and 5.  
For example, if the arrays is {1, 8, 7, 6, 0}, output must be: 8760. And if the arrays is {7, 7, 7, 6}, output must be: “no number can be formed”.

Since the number has to be divisible by 2 and 5, it has to have last digit as 0. So if the given array doesn’t contain any zero, then no solution exists.

Once a 0 is available, extract 0 from the given array. Only thing left is, the number should be is divisible by 3 and the largest of all.

The problem for finding largest divisible by 3 is discussed above...

TC : O(n)

SC : O(n)

**Select a random number from stream, with O(1) space**

Given a stream of numbers, generate a random number from the stream. You are allowed to use only O(1) space and the input is in the form of stream, so can’t store the previously seen numbers.

**1)** Initialize ‘count’ as 0, ‘count’ is used to store count of numbers seen so far in stream.  
**2)** For each number ‘x’ from stream, do following  
…..**a)** Increment ‘count’ by 1.  
…..**b)** If count is 1, set result as x, and return result.  
…..**c)** Generate a random number from 0 to ‘count-1′. Let the generated random number be i.  
…..**d)** If i is equal to ‘count – 1′, update the result as x.

**CODE:**

int selectRandom(int x)

{

    static int res;    // The resultant random number

    static int count = 0;  //Count of numbers visited so far in stream

    count++;  // increment count of numbers seen so far

    // If this is the first element from stream, return it

    if (count == 1)

        res = x;

    else

    {

        // Generate a random number from 0 to count - 1

        int i = rand() % count;

        // Replace the prev random number with new number with 1/count probability

        if (i == count - 1)

            res  = x;

    }

    return res;

}

// Driver program to test above function.

int main()

{

    int stream[] = {1, 2, 3, 4};

    int n = sizeof(stream)/sizeof(stream[0]);

    // Use a different seed value for every run.

    srand(time(NULL));

    for (int i = 0; i < n; ++i)

        printf("Random number from first %d numbers is %d \n",

                                i+1, selectRandom(stream[i]));

    return 0;

}

Auxiliary Space: O(1)

**How does this work?**  
We need to prove that every element is picked with 1/n probability where n is the number of items seen so far. For every new stream item x, we pick a random number from 0 to ‘count -1′, if the picked number is ‘count-1′, we replace the previous result with x.

To simplify proof, let us first consider the last element, the last element replaces the previously stored result with 1/n probability. So probability of getting last element as result is 1/n.

Let us now talk about second last element. When second last element processed first time, the probability that it replaced the previous result is 1/(n-1). The probability that previous result stays when nth item is considered is (n-1)/n. So probability that the second last element is picked in last iteration is [1/(n-1)] \* [(n-1)/n] which is 1/n.

Similarly, we can prove for third last element and others.

AWESOME PROOF...

**Efficient program to print all prime factors of a given number**

Given a number n, write an efficient function to print all [prime factors](http://en.wikipedia.org/wiki/Prime_factor) of n. For example, if the input number is 12, then output should be “2 2 3″. And if the input number is 315, then output should be “3 3 5 7″.

1. Following are the steps to find all prime factors.  
   While n is divisible by 2, print 2 and divide n by 2.
2. After step 1, n must be odd. Now start a loop from i = 3 to **square root** of n. While i divides n, print i and divide n by i, increment i by 2 and continue.
3. If n is a prime number and is greater than 2, then n will not become 1 by above two steps. So print n if it is greater than 2.

**Check divisibility by 7**

Given a number, check if it is divisible by 7. You are not allowed to use modulo operator, floating point arithmetic is also not allowed.

A simple method is repeated subtraction. Following is another interesting method.

Divisibility by 7 can be checked by a recursive method. A number of the form 10a + b is divisible by 7 if and only if a – 2b is divisible by 7. In other words, subtract twice the last digit from the number formed by the remaining digits. Continue to do this until a small number.

**Example:** the number 371: 37 – (2×1) = 37 – 2 = 35; 3 – (2 × 5) = 3 – 10 = -7; thus, since -7 is divisible by 7, 371 is divisible by 7.

**Generate integer from 1 to 7 with equal probability**

Given a function foo() that returns integers from 1 to 5 with equal probability, write a function that returns integers from 1 to 7 with equal probability using foo() only. Minimize the number of calls to foo() method. Also, use of any other library function is not allowed and no floating point arithmetic allowed.

**Solution:**  
We know foo() returns integers from 1 to 5. How we can ensure that integers from 1 to 7 occur with equal probability?  
If we somehow generate integers from 1 to a-multiple-of-7 (like 7, 14, 21, …) with equal probability, we can use modulo division by 7 followed by adding 1 to get the numbers from 1 to 7 with equal probability.

We can generate from 1 to 21 with equal probability using the following expression.

5\*foo() + foo() -5

Let us see how above expression can be used.  
1. For each value of first foo(), there can be 5 possible combinations for values of second foo(). So, there are total 25 combinations possible.  
2. The range of values returned by the above equation is 1 to 25, each integer occurring exactly once.  
3. If the value of the equation comes out to be less than 22, return modulo division by 7 followed by adding 1. Else, again call the method recursively. The probability of returning each integer thus becomes 1/7.

**CODE:**

int foo() // given method that returns 1 to 5 with equal probability

{

    // some code here

}

int my\_rand() // returns 1 to 7 with equal probability

{

    int i;

    i = 5\*foo() + foo() - 5;

    if (i < 22)

        return i%7 + 1;

    return my\_rand();

}

## Write a program to add two numbers in base 14

Just add the numbers in base 14 in same way we add in base 10. Add numerals of both numbers one by one from right to left. If there is a carry while adding two numerals, consider the carry for adding next numerals.

Let us consider the presentation of base 14 numbers same as hexadecimal numbers

A --> 10

B --> 11

C --> 12

D --> 13

Example:

num1 = 1 2 A

num2 = C D 3

1. Add A and 3, we get 13(D). Since 13 is smaller than

14, carry becomes 0 and resultant numeral becomes D

2. Add 2, D and carry(0). we get 15. Since 15 is greater

than 13, carry becomes 1 and resultant numeral is 15 - 14 = 1

3. Add 1, C and carry(1). we get 14. Since 14 is greater

than 13, carry becomes 1 and resultant numeral is 14 - 14 = 0

Finally, there is a carry, so 1 is added as leftmost numeral and the result becomes

101D

**Notes:**  
Above approach can be used to add numbers in any base. We don’t have to do string operations if base is smaller than 10.

**Make a fair coin from a biased coin**

You are given a function foo() that represents a biased coin. When foo() is called, it returns 0 with 60% probability, and 1 with 40% probability. Write a new function that returns 0 and 1 with 50% probability each. Your function should use only foo(), no other library method.

**Solution:**  
We know foo() returns 0 with 60% probability. How can we ensure that 0 and 1 are returned with 50% probability?  
The solution is similar to [this](http://www.geeksforgeeks.org/archives/22539) post. If we can somehow get two cases with equal probability, then we are done. We call foo() two times. Both calls will return 0 with 60% probability. So the two pairs (0, 1) and (1, 0) will be generated with equal probability from two calls of foo(). Let us see how.

**(0, 1):** The probability to get 0 followed by 1 from two calls of foo() = 0.6 \* 0.4 = 0.24  
**(1, 0):** The probability to get 1 followed by 0 from two calls of foo() = 0.4 \* 0.6 = 0.24

*So the two cases appear with equal probability. The idea is to return consider only the above two cases, return 0 in one case, return 1 in other case. For other cases [(0, 0) and (1, 1)], recur until you end up in any of the above two cases.*

int foo() //given method that returns 0 with 60% probability and 1 with 40%

{

    // some code here

}

// returns both 0 and 1 with 50% probability

int my\_fun()

{

    int val1 = foo();

    int val2 = foo();

    if (val1 == 0 && val2 == 1)

        return 0;   // Will reach here with 0.24 probability

    if (val1 == 1 && val2 == 0)

        return 1;   // // Will reach here with 0.24 probability

    return my\_fun();  // will reach here with (1 - 0.24 - 0.24) probability

}