```
# to ensure fixed result, hash needs to be seeded by running:
# PYTHONHASHSEED=0 python3.9 code.py

'''
TASK:
Write a random sampling without replacement algo.
This challenge is much more substantial that it may first appear.
-At least once n (the sample size) and N (the population size) get large.

KEY INSIGHT:
The optimal solution (in terms of runtime complexity) depends on:
1) size of the population N to sample from.
2) size of the sample n to draw.
3) n/N ratio
4) if N and n fit into memory.

ALGO OPTIONS:
A) For small n/N where n fits in memory, the best approach is
to keep drawing random samples until n is reached while tracking and skipping
items that have been sampled. This algo can run in Ot(n) time.
However, a large n/N means that many loops are needed at the end
of the algo. E.g. If N = 1000, n = 999 and 998 items have been sampled,
randomly finding one of the 2/1000 that would get us out of the loop may take a
while!

B) If n/N is large and N fits in memory, the best approach is
to create a list containing N, shuffle it (using fisher_yates) and pop
n elements from the end.
This approach takes Ot(N) which can be bad for large N.

C) If n/N is large and N fits in memory, we can also take the approach of
splitting the problem into subproblems.
The advantage of splitting N and n is that we can use algo A)
even if the original n/ N was small.
e.g. If N = 1000 and n = 999 and 998 items have been sampled,
randomly finding one of the 2/1000 that would close the loop may take a
while. But if we split down to something in the domain of N = 5 and n = 4,
then getting the last item has a chance of 2/5 which is much
faster than 2/ 1000.

The challenge here is to ensure
that all distributions of n are possible.
E.g. if N = 10 and n = 2 is split across 2 compute instances e.g. allocating
half of N to each instance, we can NOT simply allocate 1 sample to each
instance.

Consider:
n = 2
N = 0,1,2,3,4,5,6,7,8,9

n1 = 1
N1 = 0,1,2,3,4

n2 = 1
N2 = 5,6,7,8,9

Here, we would not be able to sample 8 and 9. So, we
need to ensure that by random chance all of the the
following sample allocations across the above split
of N into N1 and N2 are possible:
n1: 2, n2 = 0,
n1: 1, n2 = 1 (the most likely outcome)
n1: 0, n2 = 2.

To this end we randomly select from the hypergeometric distribution
of (n, N - n, N/2). So asking the question of how many wins we get if
we have a jar with n wins, N-n losses, and draw N/2 items.
The resulting number x becomes n1 while n2 becomes n-x.
```

D) If memory is an issue, one approach is to divide up the population N and the
sample n across multiple workers. One way to do this is by using the
hypergeometric distribution approach similar to what is described under C)
above.

If the following only fits into memory:
N = 1000 and n = 100
after split N in half 7 times (to distribute across 8 workers),
we can do so recursively and distribute samples across splits
by randomly sampling the hypergeometric distribution to ensure that any
distribution of samples across workers is possible.

Once the population is split, algo C is used on each worker.

Algo D is use on i parallel workers. Challenge is to split the
data onto workers consistently. So, the same random deviates need to be used on
every worker. Once the data is subset for a given worker algo C takes over.
Splitting under algo C needs to use different random
deviates on each worker to ensure that sampling is independent.
I ensure this by seeding Algo D the same way on every worker and once algo C
takes over, I pass a seed that is a function of the worker number i.

ATTRIBUTION (!):
I coded A, B and especially C and D by translating c code following:
https://arxiv.org/pdf/1610.05141.pdf and
https://github.com/lorenzhs/sampling.

LIMITATIONS OF MY SOLUTION:
I was not able to get a DASK parallelization on my quad core
laptop to run faster than at least one of the sequential option.
I suspect that python/ DASK and a tiny cluster of 1 quad core is
not suitable to demo a parallelization speedup.

HYPERGEOMETRIC SAMPLING
I am using np.random.hypergeometric to do the hypergeometric sampling.
I could also do hypergeometric sampling from scratch using the sampling without
replacement options coded here.

E.g. sampling n = 2 out of a population with
3 wins and 2 losses is a sampling without replacement of 2 = n, N = 5.
Picture this array that we sample from:
indices: 0, 1, 2, 3, 4
values: 0, 0, 1, 1, 1
One samples 2 indicines randomly without replacement and then counts the
number of times the sampled indices are >= 2 to count the wins (when using
algo A). If shuffling the array (algo B), one can sum the samples to count
the wins.

MORE ALGOS
There are more algos to choose from than coded here.
See e.g. https://arxiv.org/pdf/1610.05141.pdf

MOST IMPORTANTLY:
Have fun :)

mark.broich@gmail.com; Spring 2022.
'''

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random
import time
from collections import defaultdict
from dask.distributed import Client


# Classes
class SamplingMethods:
    '''Class with algos A, B and C (randoms sampling with
    replacement methods).
    main_method is either "divide", "hash" or "fisher_yates".
    hash_seed sets the random see to allow reproducability of results.
    If main_method is "divide", sub_method of either "hash" or "fisher_yates"
    need to be specified and used.
    For main_method "divide" base_size determines when recursive splitting
    stops and base (aka "sub_method" algorithms) is used from there on.
    '''
    def __init__(self, main_method: str = 'divide', sub_method: str = 'hash',
                 base_size: int = 5, hash_seed: int = 1):
        self.main_method = main_method
        self.sub_method = sub_method
        self.hash_seed = hash_seed
        self.base_size = base_size

    def sampling(self, N: int, n: int, offset: int = 0) -> list:
        '''Main methoid to take and returns n random samples w/o replacement
        from the range 0 to N + offset and results result using algo
        A, B, or C as defined during class init.
        h is the random seed'''

        if self.hash_seed == -1:
            h = self.hash_seed
        else:
            h = hash(self.hash_seed)
        if self.main_method == 'hash':
            # Algo A)
            return self.sampling_wo_replacement_hash(N, n, offset, h)
        elif self.main_method == 'fisher_yates':
            # Algo B
            return self.n_rand_ints_without_replacement_shuffel_list(N, n,
                                                     offset, h)
        elif self.main_method == 'divide':
            # Algo C
            return self.seq_divide_sampling(N, n, offset, h)

    # Algo A
    def sampling_wo_replacement_hash(self, N: int, n: int,
                                     offset: int = 0, h=-1) -> list:
        '''Take and returns n random samples w/o replacement
        from the range 0 to N + offset. h is the random seed'''
        # set seeds if h is passed(only happends during first call)
        if h != -1:
            random.seed(h)
            np.random.seed(h)
        seen = set()
        result = [''] * n
        i = 0
        while i < n:
            x = self.randint(0, N)
            if x not in seen:
                seen.add(x)
                result[i] = x + offset
                i += 1
        return result
```

```
# Algo B
def n_rand_ints_without_replacement_shuffel_list(self, N: int, n: int,
                                        offset: int = 0, h:
                                        int = -1) -> list:
    '''Take and returns n random samples w/o replacement
    from the range 0 to N + offset. h is the random seed'''
    # set seeds if h is passed(only happends during first call)
    if h != -1:
        random.seed(h)
        np.random.seed(h)
    nums = [i + offset for i in range(0, N)]
    if len(nums) < n:
        return []
    self.shuffel_list(nums)
    return nums[0:n]

# Algo C
def seq_divide_sampling(self, N, n, offset=0, h=-1) -> list:
    '''Take and returns n random samples w/o replacement
    from the range 0 to N + offset. h is the random seed.'''
    # set seeds if h is passed(only happends during first call)
    if h != -1:
        random.seed(h)
        np.random.seed(h)
    if n <= self.base_size:
        if self.sub_method == 'hash':
            # Allocate hash table for base case
            return self.sampling_wo_replacement_hash(N, n, offset, h)
        elif self.sub_method == 'fisher_yates':
            return self.n_rand_ints_without_replacement_shuffel_list(
                                            N, n, offset, h)
    N_split = int(N / 2)  # N split
    x = np.random.hypergeometric(n, N - n, N_split)
    result1 = self.seq_divide_sampling(N_split, x, offset)
    result2 = self.seq_divide_sampling(N - N_split, n - x, offset
                                + N_split)
    return result1 + result2

def randint(self, mini: int, N: int) -> int:
    ''' returns random int between mini and N inclusive'''
    return int(random.random() * (N - mini) + mini)

def shuffel_list(self, nums: list) -> None:
    '''return a shuffeled list'''
    # using Fisher–Yates shuffle Algorithm
    for i in range(len(nums)-1, 0, -1):
        # Pick a random index from 0 to i
        j = self.randint(0, i)
        nums[i], nums[j] = nums[j], nums[i]
```

```
class ParallelDivideSampling(SamplingMethods):
    ''''Class to run algo D:
    So, recursively splits population and sampels for i'th workers
    and then launches algo C on the partition.
    Inherits SamplingMethods to have access to algo A, B, C.
    --
    N and n is split using the same random seed on every worker to ensure
    constiant allocation of worker-specific data.
    Once the data is "worker specifc", algo C is launched with a different
    random seed per worker to ensure randomness between workers'''
    def __init__(self, universe_size, sub_method: str = 'hash',
                 base_size: int = 5, seed: int = 1, wkr_cnt: int = 1):
        self.hash_seed = seed
        self.main_method = 'divide'
        self.sub_method = sub_method
        self.base_size = base_size
        # Compute input distribution
        self.rem = universe_size % wkr_cnt
        self.div = int(universe_size / wkr_cnt)

    def sample(self, n: int, j: int = 1, k: int = 1, i: int = 1, offset=0):
        ''''Runs algo D to take and returns n random samples w/o replacement
        from the range 0 to N + offset.
        Purpose is to break up the universe across the workers the same way
        way on each worker before passing to algo C with a per-worker
        random seed.
        h is the random seed.
        n is the sample size.
        j is the current lower worker index.
        k is the current upper worker index.
        i is the worker number.
        '''
        if(j - k == 0):
            # Allocate hash table for base case
            h = hash(self.hash_seed + i)
            return self.seq_divide_sampling(self.N(i+1) - self.N(i), n,
                                            offset, h)

        m = int((j + k) / 2)
        N_split = self.N(m) - self.N(j-1)

        if self.hash_seed == -1:
            self.hash_seed = self.randint(0, 2**32 - 1 - j - k)
        h = hash(self.hash_seed + j + k)
        np.random.seed(h)
        x = np.random.hypergeometric(n, self.N(k) - self.N(j-1) - n, N_split)

        if (i < m):
            return self.sample(x, j, m, i, offset)
        else:
            return self.sample(n-x, m+1, k, i, offset + N_split)

    def N(self, i):  # i is worker i
        '''helper that returns per of N to which worker i it is given'''
        return i * self.div + min(i, self.rem)


# helper functions
def flattern(result: list, n: int) -> list:
    """Retrun flatterend version of a list"""
    result_flat = [''] * n
    idx = 0
    for part in result:
        for p in part:
            result_flat[idx] = p
            idx += 1
    return result_flat
```

```
def make_set(result: list) -> set:
    """turn a list into a set"""
    uni = set()
    for i in result:
        uni.add(i)
    return uni


def run_in_series(pds, wkr_cnt: int, n: int, j: int = 1,
                  offset: int = 0) -> list:
    '''runs sample method of instance of ParallelDivideSampling for one worker at
    in series appends the result and returns it'''
    result = []
    for i in range(wkr_cnt):
        result = result + (pds.sample(n, j=j, k=wkr_cnt, i=i, offset=offset))
    return result


def make_input_params(n: int, j: int, k: int, i: int, wkr_cnt: int):
    '''returns input parameters n, j, i, worker count as a pd dataframe'''
    data = {'n': [n for i in range(wkr_cnt)],
            'j': [1 for i in range(wkr_cnt)],
            'k': [wkr_cnt for i in range(wkr_cnt)],
            'i': [i for i in range(wkr_cnt)]}
    return pd.DataFrame(data)


def get_result_for_x_runs(instance, x: int = 1000, universe_size: int = 100,
                          n: int = 10, wkr_cnt: int = 4, algo_d: bool = False
                          ) -> dict:
    '''Run function x times and returns count of sampeled
    numbers in dict sorted by sampled values'''
    instance.hash_seed = -1
    result_cnt = defaultdict(int)
    for i in range(x):
        if algo_d:
            instance.hash_seed = -1
            result = run_in_series(instance, wkr_cnt, n)
        else:
            result = instance.sampling(universe_size, n)
        for r in result:
            result_cnt[r] += 1
    return sorted(result_cnt.items())


def plot_value_occurance(result_cnt: dict, file_name: str = 'plot') -> None:
    '''Plots content of dict with keys on x and
    occurance count on y'''
    x = []
    y = []
    for k, v in result_cnt:
        x.append(k)
        y.append(v)
    plt.bar(x, y, color='green',
            width=0.5)
    plt.xlabel("value")
    plt.ylabel("number of times sampled")
    plt.title(file_name)
    plt.suptitle("Check is range 0 - 99 was randomly sampeled")
    # plt.show()
    plt.savefig(file_name + '.png')
```

```
def functional_tests():
    '''functional_tests tests classes and their methods and print results
    to screen'''
    seed = 1

    print('--Test results of algos A, B, C. Sampling 10 of 10')
    universe_size = 10
    n = 10
    ms = SamplingMethods(main_method='hash', hash_seed=seed)
    print('algo A:', sorted(ms.sampling(N=universe_size, n=n)) ==
        [i for i in range(n)])

    ms = SamplingMethods(main_method='fisher_yates', hash_seed=seed)
    print('algo B:', sorted(ms.sampling(N=universe_size, n=n)) ==
        [i for i in range(n)])

    ms = SamplingMethods(main_method='divide', sub_method='hash',
                         hash_seed=seed)
    print('algo C:', sorted(ms.sampling(N=universe_size, n=n)) ==
        [i for i in range(n)])

    print('--Test results of algos A, B, C. Sampling 10 of 10 with offset')
    offset = 10
    ms = SamplingMethods(main_method='hash', hash_seed=seed)
    print('algo A w offset:', sorted(ms.sampling(N=universe_size, n=n,
        offset=offset)) == [i for i in range(offset, n + offset)])

    ms = SamplingMethods(main_method='fisher_yates', hash_seed=seed)
    print('algo B w offset:', sorted(ms.sampling(N=universe_size, n=n,
        offset=offset)) == [i for i in range(offset, n + offset)])

    ms = SamplingMethods(main_method='divide', sub_method='hash',
                         hash_seed=seed)
    print('algo C w offset:', sorted(ms.sampling(N=universe_size, n=n,
        offset=offset)) == [i for i in range(offset, n + offset)])

    print('--Test results of ParallelDivideSampling class running Algo C '
        'in series with algo A, B as sub methods. Sampling 10 of 10.')
    sub_method = 'hash'
    base_size = 3
    wkr_cnt = 3
    pds = ParallelDivideSampling(universe_size, sub_method, base_size,
                                 seed, wkr_cnt=wkr_cnt)

    result = run_in_series(pds, wkr_cnt, n)
    print('Algo C in series with algo A as sub methods wkr_cnt = 3, len:',
        len(result) == n)
    print('Algo C in series with algo A as sub methods wkr_cnt = 3:',
        sorted(result) == [i for i in range(universe_size)])

    sub_method = 'fisher_yates'
    pds = ParallelDivideSampling(universe_size, sub_method, base_size,
                                 seed, wkr_cnt=wkr_cnt)
    result = run_in_series(pds, wkr_cnt, n)
    print('Algo C in series with algo B as sub methods wkr_cnt = 3, len:',
        len(result) == n)
    print('Algo C in series with algo B as sub methods wkr_cnt = 3:',
        sorted(result) == [i for i in range(universe_size)])

    print('--Different wkr_cnt')
    universe_size = 10
    wkr_cnt = 7
    pds = ParallelDivideSampling(universe_size, sub_method, base_size,
                                 seed, wkr_cnt=wkr_cnt)
    result = run_in_series(pds, wkr_cnt, n)
    print('Algo C in series with algo B as sub methods wkr_cnt = 7, len:',
        len(result) == n)
    print('Algo C in series with algo B as sub methods wkr_cnt = 7:',
        sorted(result) == [i for i in range(universe_size)])
```

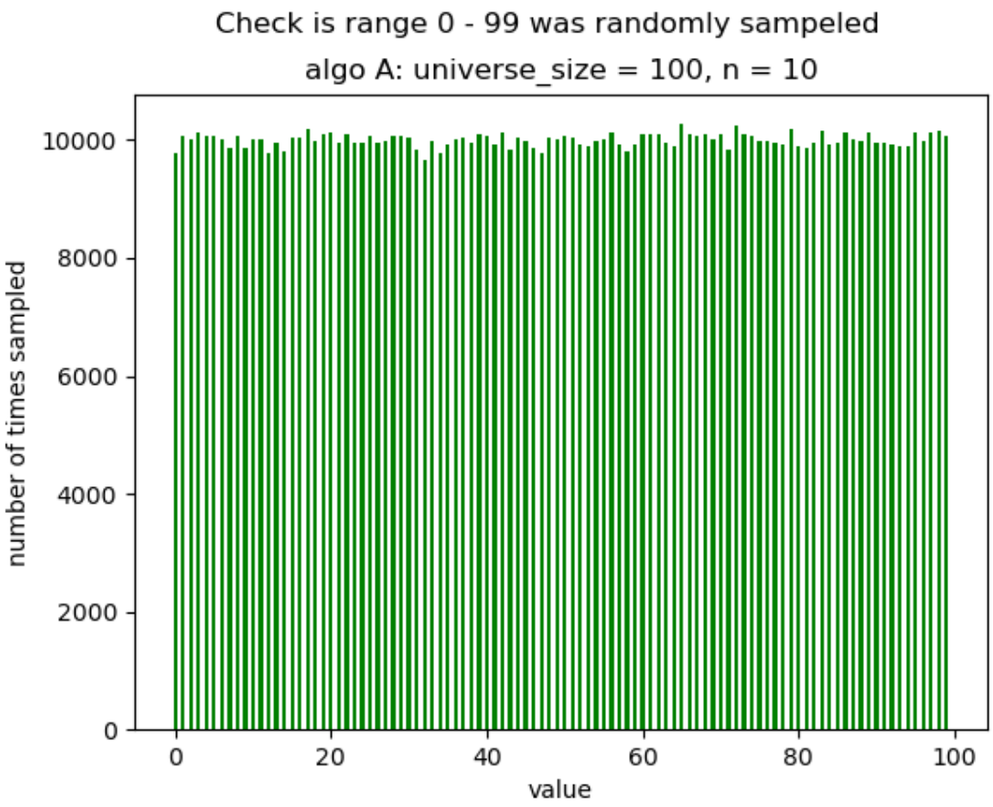
```
print('--Test results of ParallelDivideSampling class running Algo C '
    'in series with algo A, B as sub methods Sampling 10 of 10 with offset.')
offset = 10
sub_method = 'hash'
base_size = 3
wkr_cnt = 3
pds = ParallelDivideSampling(universe_size, sub_method, base_size,
                        seed, wkr_cnt=wkr_cnt)

result = run_in_series(pds, wkr_cnt, n, offset=offset)
print('Algo C in series with algo A as sub methods. wkr_cnt = 3, offset '
    'len:', len(result) == n)
print('Algo C in series with algo A as sub methods. wkr_cnt = 3, offset:',
    sorted(result) == [i for i in range(offset, n + offset)])

sub_method = 'fisher_yates'
pds = ParallelDivideSampling(universe_size, sub_method, base_size,
                        seed, wkr_cnt=wkr_cnt)
result = run_in_series(pds, wkr_cnt, n, offset=offset)
print('Algo C in series with algo B as sub methods. wkr_cnt = 3, offset, '
    'len:', len(result) == n)
print('Algo C in series with algo B as sub methods. wkr_cnt = 3, offset:',
    sorted(result) == [i for i in range(offset, n + offset)])

print('--Different wkr_cnt')
universe_size = 10
wkr_cnt = 7
pds = ParallelDivideSampling(universe_size, sub_method, base_size,
                        seed, wkr_cnt=wkr_cnt)
result = run_in_series(pds, wkr_cnt, n, offset=offset)
print('Algo C in series with algo B as sub methods. wkr_cnt = 7, offset, '
    'len:', len(result) == n)
print('Algo C in series with algo B as sub methods. wkr_cnt = 7, offset:',
    sorted(result) == [i for i in range(offset, n + offset)])

print('--Test results of ParallelDivideSampling class running Algo C in '
    'series with algo A, B as sub methods. changing universe_size, '
    'base_size, n and wkr_cnt')
sub_method = 'hash'
universe_size = 100
base_size = 10
pds = ParallelDivideSampling(universe_size, sub_method, base_size,
                        seed, wkr_cnt=wkr_cnt)
n = 50
result = run_in_series(pds, wkr_cnt, n)
print('Algo C in series with algo B as sub methods. Changing '
    'universe_size, base_size, n and wkr_cnt. len:',
    len(result) == n)
print('Algo C in series with algo B as sub methods. Changing '
    'universe_size, base_size, n and wkr_cnt. unique:',
    len(result) == len(make_set(result)))

universe_size = 10000
wkr_cnt = 21
pds = ParallelDivideSampling(universe_size, sub_method, base_size,
                        seed, wkr_cnt=wkr_cnt)
n = 1000
result = run_in_series(pds, wkr_cnt, n)
print('Algo C in series with algo B as sub methods. Changing '
    'universe_size, base_size, n and wkr_cnt. len:',
    len(result) == n)
print('Algo C in series with algo B as sub methods. Changing '
    'universe_size, base_size, n and wkr_cnt. unique:', len(result)
    == len(make_set(result)))
print('--All bool tests done----------')
```

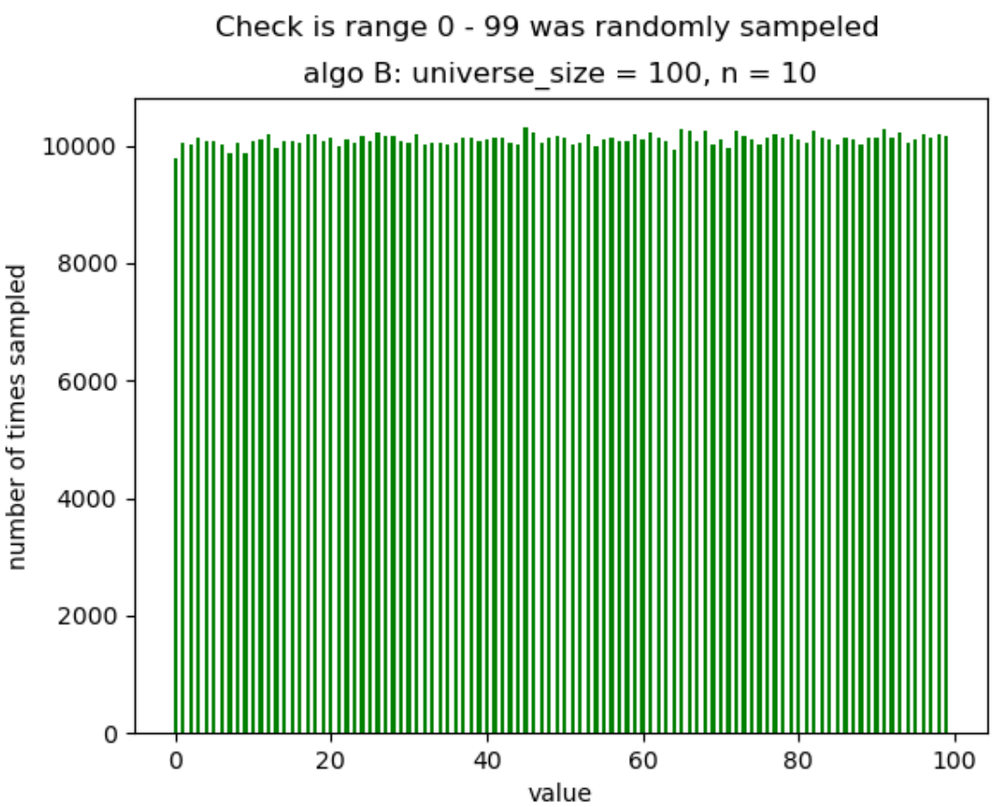
```
def plotting_tests():
    '''Plots trial runs to allow evaluation of randomness'''
    print('--Check for randomness by plotting histrogram of 10 numbers '
          'drawn 100k times out of 1 to 100 population using algo A,B,C,D')
    print('=> Algos produce random numbers if 1-100 occures ~equally often '
          'in the plots')
    seed = 1
    universe_size = 100
    n = 10
    simulation_runs = 100000
    ms = SamplingMethods(main_method='hash', hash_seed=seed)
    result_cnt = get_result_for_x_runs(instance=ms, x=simulation_runs, n=n,
                                universe_size=universe_size)
    plot_value_occurance(result_cnt, 'algo A: universe_size = 100, n = 10')

    ms = SamplingMethods(main_method='fisher_yates', hash_seed=seed)
    result_cnt = get_result_for_x_runs(instance=ms, x=simulation_runs, n=n,
                                universe_size=universe_size)
    plot_value_occurance(result_cnt, 'algo B: universe_size = 100, n = 10')

    ms = SamplingMethods(main_method='divide', sub_method='hash',
                    hash_seed=seed)
    result_cnt = get_result_for_x_runs(instance=ms, x=simulation_runs, n=n,
                                universe_size=universe_size)
    plot_value_occurance(result_cnt, 'algo C: universe_size = 100, n = 10')

    sub_method = 'hash'
    base_size = 3
    wkr_cnt = 4
    pds = ParallelDivideSampling(universe_size=universe_size,
                            sub_method=sub_method,
                            base_size=base_size,
                            seed=seed, wkr_cnt=wkr_cnt)

    result_cnt = get_result_for_x_runs(pds, x=simulation_runs,
                                n=n, wkr_cnt=wkr_cnt, algo_d=True)
    plot_value_occurance(result_cnt, 'algo D: universe_size = 100, n = 10')
    print('--All plotting tests done----------')


def runtime_tests():
    '''Times algos A, B, C, D for fixe sample and universe.
    Algo D is run on workers in series and, using dask, in parallel. '''
    seed = 1
    universe_size = 100000
    n = 50000
    base_size = 10000

    print('--Test of runtime algos A, B, C')
    t1 = time.time()
    ms = SamplingMethods(main_method='hash', base_size=base_size,
                    hash_seed=seed)
    ms.sampling(N=universe_size, n=n)
    print('Algos B. Secs:', round(time.time() - t1, 4))

    t1 = time.time()
    ms = SamplingMethods(main_method='fisher_yates', base_size=base_size,
                    hash_seed=seed)
    ms.sampling(N=universe_size, n=n)
    print('Algos B. Secs:', round(time.time() - t1, 4))

    t1 = time.time()
    ms = SamplingMethods(main_method='divide', sub_method='hash',
                    base_size=base_size, hash_seed=seed)
    ms.sampling(N=universe_size, n=n)
    print('Algo C with algo A as submethod. Secs:', round(time.time() - t1, 4))
```

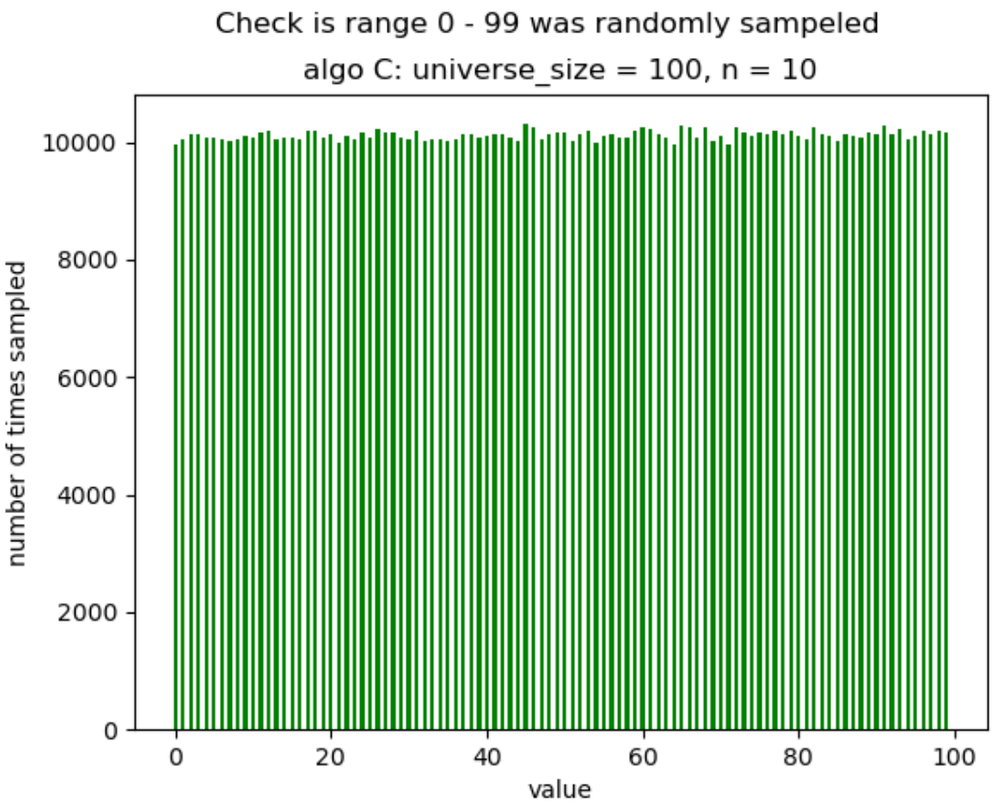
```
t1 = time.time()
ms = SamplingMethods(main_method='divide', sub_method='fisher_yates',
                base_size=base_size, hash_seed=seed)
ms.sampling(N=universe_size, n=n)
print('Algo C with algo B as submethod. Secs:', round(time.time() - t1, 4))

sub_method = 'hash'
# sub_method = 'fisher_yates'
wkr_cnt = 4
thread_cnt = 1
base_size = 1024
pds = ParallelDivideSampling(universe_size, sub_method, base_size,
                        seed, wkr_cnt=wkr_cnt)

t1 = time.time()
result = []
for i in range(wkr_cnt):
   result = result + (pds.sample(n=n, j=1, k=wkr_cnt, i=i))
print('ParallelDivideSampling class running algo C in '
    'series algo A as submethod. Secs:', round(time.time() - t1, 4))
print('--Test results of algos C run in series with algo A as submethod.')
print(len(result) == n)
print(len(result) == len(make_set(result)))

client = Client(threads_per_worker=thread_cnt, n_workers=wkr_cnt)
t1 = time.time()
input_params = make_input_params(n, 1, wkr_cnt, i, wkr_cnt)
futures = client.map(pds.sample,
                input_params['n'].values,
                input_params['j'].values,
                input_params['k'].values,
                input_params['i'].values)
result = client.gather(futures)
result = flattern(result, n)
print('ParallelDivideSampling class running algo D in parallel with algos '
    'C and A as submethod. Secs:', round(time.time() - t1, 4))

print('--Test results of ParallelDivideSampling class running algos D '
    'run in parallel with algo A as submethod.')
print(len(result) == n)
print(len(result) == len(make_set(result)))

print('--Test results of ParallelDivideSampling class running algos D '
    'run in parallel with algo A as submethod. '
    'Different wkr_cnt and sampling 10 of 10')
universe_size = 10
base_size = 3
seed = 1
wkr_cnt = 7
pds = ParallelDivideSampling(universe_size, sub_method, base_size,
                        seed, wkr_cnt=wkr_cnt)
n = 10
input_params = make_input_params(n, 1, wkr_cnt, i, wkr_cnt)
futures = client.map(pds.sample,
                input_params['n'].values,
                input_params['j'].values,
                input_params['k'].values,
                input_params['i'].values)
result = client.gather(futures)
result = flattern(result, n)
print(len(result) == n)
print(sorted(result) == [s for s in range(universe_size)])
client.shutdown()
print('--All runtime tests done----------')
```

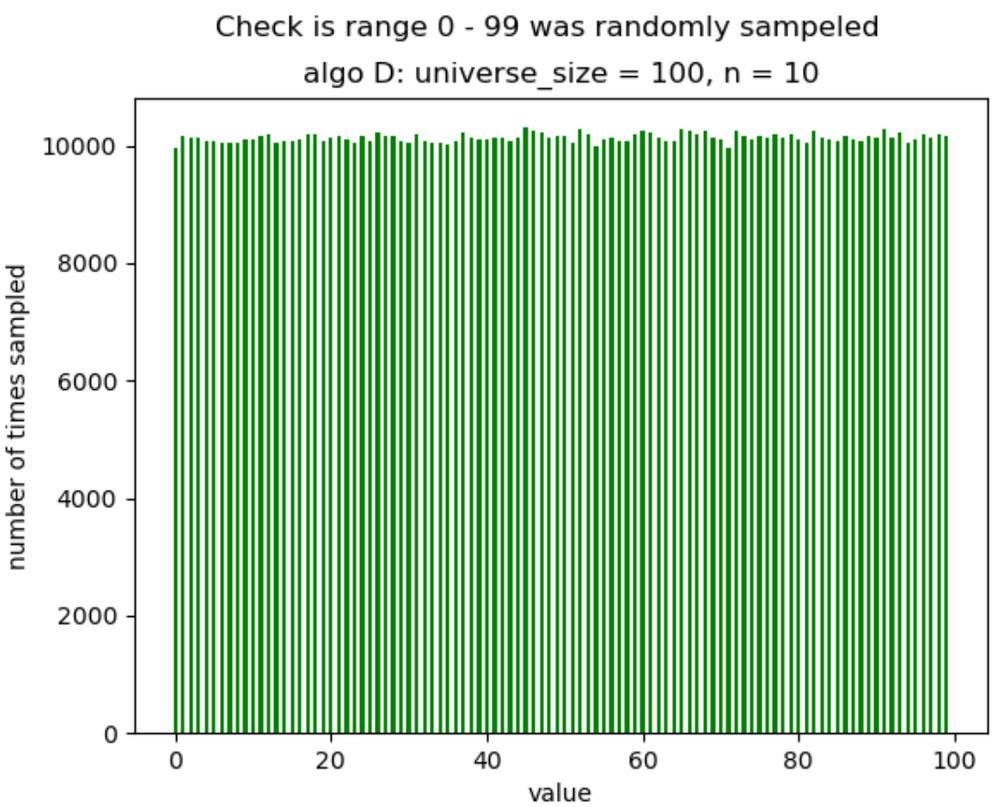
```python
if __name__ == "__main__":
    # run the test
    functional_tests()
    plotting_tests()
    runtime_tests()
```

Test results
--Test results of algos A, B, C. Sampling 10 of 10
algo A: True
algo B: True
algo C: True
--Test results of algos A, B, C. Sampling 10 of 10 with offset
algo A w offset: True
algo B w offset: True
algo C w offset: True
--Test results of ParallelDivideSampling class running Algo C in series with algo A, B as sub methods. Sampling 10 of 10.
Algo C in series with algo A as sub methods wkr_cnt = 3, len: True
Algo C in series with algo A as sub methods wkr_cnt = 3: True
Algo C in series with algo B as sub methods wkr_cnt = 3, len: True
Algo C in series with algo B as sub methods wkr_cnt = 3: True
--Different wkr_cnt
Algo C in series with algo B as sub methods wkr_cnt = 7, len: True
Algo C in series with algo B as sub methods wkr_cnt = 7: True
--Test results of ParallelDivideSampling class running Algo C in series with algo A, B as sub methods Sampling 10 of 10 with offset.
Algo C in series with algo A as sub methods. wkr_cnt = 3, offset len: True
Algo C in series with algo A as sub methods. wkr_cnt = 3, offset: True
Algo C in series with algo B as sub methods. wkr_cnt = 3, offset, len: True
Algo C in series with algo B as sub methods. wkr_cnt = 3, offset: True
--Different wkr_cnt
Algo C in series with algo B as sub methods. wkr_cnt = 7, offset, len: True
Algo C in series with algo B as sub methods. wkr_cnt = 7, offset: True
--Test results of ParallelDivideSampling class running Algo C in series with algo A, B as sub methods. changing universe_size, base_size, n and wkr_cnt
Algo C in series with algo B as sub methods. Changing universe_size, base_size, n and wkr_cnt. len: True
Algo C in series with algo B as sub methods. Changing universe_size, base_size, n and wkr_cnt. unique: True
Algo C in series with algo B as sub methods. Changing universe_size, base_size, n and wkr_cnt. len: True
Algo C in series with algo B as sub methods. Changing universe_size, base_size, n and wkr_cnt. unique: True
--All bool tests done----------
--Check for randomness by plotting histrogram of 10 numbers drawn 100k times out of 1 to 100 population using algo A,B,C,D
=> Algos produce random numbers if 1-100 occures ~equally often in the plots
--All plotting tests done----------
--Test of runtime algos A, B, C
Algos A. Secs: 0.0645
Algos B. Secs: 0.0819
Algo C with algo A as submethod. Secs: 0.0458
Algo C with algo B as submethod. Secs: 0.0864
ParallelDivideSampling class running algo C in series algo A as submethod. Secs: 0.0484
--Test results of algos C run in series with algo A as submethod.
True
True
ParallelDivideSampling class running algo D in parallel with algos C and A as submethod. Secs: 1.2215
--Test results of ParallelDivideSampling class running algos D run in parallel with algo A as submethod.
True
True
--Test results of ParallelDivideSampling class running algos D run in parallel with algo A as submethod. Different wkr_cnt and sampling 10 of 10
True
True
--All runtime tests done----------


Check is range 0 - 99 was randomly sampeled
algo A: universe_size = 100, n = 10
number of times sampled
value
0
2000
4000
6000
8000
10000
0
20
40
60
80
100


Check is range 0 - 99 was randomly sampeled
algo B: universe_size = 100, n = 10
number of times sampled
value
0
2000
4000
6000
8000
10000
0
20
40
60
80
100


Check is range 0 - 99 was randomly sampeled
algo C: universe_size = 100, n = 10
number of times sampled
value
0
2000
4000
6000
8000
10000
0
20
40
60
80
100


Check is range 0 - 99 was randomly sampeled
algo D: universe_size = 100, n = 10
number of times sampled
value
0
2000
4000
6000
8000
10000
0
20
40
60
80
100