CSC211—Problem Solving, Algorithms and Programming Shuffle numbers in $[1..N]$ in three different ways. Practical 1 Term 1 9/11 February 2026

Submit today before: 17h20, February 2026

Please follow the instructions carefully. You will receive marks for everything done correctly that is submitted before you leave the practical *today.*

The exact instructions on the mechanics of creating your working directory and using git are as clear as I can make them. You need a place to keep your work for this course, separate from all your other computing. We want you to use **git**, and assume that you already know how to use use it.

1. **Create a working directory and do a git init before doing anything else.**

   (a) I am a long term Linux user, and cannot give you all the details for running your work on Windows, so I will just explain what I want and will fill in the minute details of how to go about submitting by next week. All your need to do after each practical is to submit all your archived work each week. You need to do regular updates of your work with git. Do the updates

   (b) The next practical will be done under the directory 12practical, and then 13practical, etc.

   (c) Your work must somehow be uploaded to iKhamva where I can look at it.

   (d) We will mark this weeks work as soon as possible and give you feedbakc. The automatic marking software is not yet in place, we are still working on it, but upload it to iKamva *today.*

2. **Implement** three different ways to *shuffle* N numbers in the range $[1..N]$ in your directory `A2practical`. Called the slow method `slowshuffle`, the biased method `biasedshuffle` and the unbiasedmethod `shuffle`.

   (a) *A very bad shuffling method*: DO NOT IMPLEMENT THIS, BUT READ THIS TO UNDERSTAND `slowshuffle`, consider the explanation first and write the program in the improved way. The very bad method generates random numbers from 1–N using `r = 1+(int)(Math.random()*N)`[1] and adds `r` to the end of an array called, `shuffled[]`, after checking that it is not already in the list. The numbers appear in the order that they are generated. The method will run slower and *slower* as it fills up because the of the time it takes to check that the number is not in the list. A very slow way of implementing this is by searching through the list until a duplicate is found and then generating another `r` and testing it in turn. The method can stop when $N-1$ numbers have been found. How is the last number found quite efficiently?

   `slowshuffle`: *Improved, very bad method*, that you must implement:
   The way to improve the very bad method into one that is slightly faster is by keeping an extra array, other than the list of random numbers called `shuffled[]`, in the order that they are found. The other array, `isNotPresent[]`, is initially

[1]This yields an integer that lies in the closed integer interval $[1..N]$

filled with `True` values. The meaning read into `isNotPresent[r] == True` is that the specific value of `r` has not yet been calculated. If, however, `isNotPresent[r] == False` then r is already present as one of the values of `shuffled[]`.

Each time a potential `r` is calculated, check if it is already in the list and if it is not in the list put it into the shuffled list. All this must be done in a while loop that fills up the array `shuffled[]`. The following is a hint in pseudo code of how and when to enter a new vlue of `r`—it reflects the idea of what you need to do, and is not necessarily absolutely correct.

```
if isNotPresent[r]:
   i += 1
   shuffled[i] = r
   isNotPresent[r] = False
else:
   r = int(random()*N) + 1
```

[2] This must simply be repeated until the second last element is entered. The last `r` need not be *generated* because it is already known—it is the index of the only element in the array `isNotPresent[]` that is still `True`. This is a simple piece of coding which you also have to program. Once you get this to work move on to implement the efficient methods.

(b) shufflebiased *An efficient method for shuffling*: First set up an array: `shuffled[]` and initialize the array so that `shuffled[1]=1`; `shuffled[2]=2`; ..., `shuffled[N]=N`, next get N random numbers $\texttt{r}_i \in \{1, 2, \ldots, N\}$ `for` $i = 1, 2, \ldots, N$ and for each such number i swap the values of `shuffled[`$\texttt{r}_i$`]` and `shuffled[i]`—after the N-th swap the list `shuffled[]` will contain the numbers in the range 1–N in random order. How many swaps were there? Beware: although this method is efficient if it is not done correctly it yields a biased shuffle. The correct way to do the shuffling is to adjust the creation of the next element to swap so that it is selected out of those numbers that have not been selected before.

Note on unbiased shuffling—Do not implement more than two shuffling methods To get your method to work correctly, without bias, think of how you would program 'extracting numbered balls one-by-one from a jar'. Suppose there there are `N` balls in the jar numbered 1, 2, ... N and suppose that these numbers are stored in an array created with

```
B = [b+1 for b in range(N)]
```

1. Set `b` to zero.
2. Repeat the following process until the jar is empty.
   2.1 'Remove' one ball at random from the jar by generating a random number `r` in the range `[i..N-1]` swap this 'ball' at `B[b]` with the ball `B[r]`.
   2.2 Set `b` to `b + 1`.

(c) Try to prove the subtle difference between this method and the previous one by shuffling the list `B`, with `N = 3` exactly 60000 times and counting the number of times that each of the triples `[1,2,3]`, `[1,3,2]`, `[2,1,3]`, `[2,3,1]`, `[3,1,2]`, `[3,2,1]` appears. This must be done for both 'good' shuffling methods.

[2]The suggested code could obviously have used `shuffled.append(r)` to add the new random number `r`, instead of entering `r` directly into the `i`-th position.

(d) If the method is unbiased then the number of appearances of each triple should be close to 10000 in all six cases. What do you observe?

Note that you may use a *dictionary* as follows to count the appearances of each triple: if `B = [1, 3, 2]`, create the string `"132"` and use this as an ‘index’ or ‘key’ in the dictionary `D`.

The dictionary `D`, much like a list, is created by putting `D = {}`. To insert an item into the dictionary given `key = "132"`, simply

```python
if key not in D:
  D[key] = 1
else:
  D[key] += 1
```

This will initialize `D[key]` to `1` if it has never been encountered before, and will add a `1` to the dictionary entry for `key` otherwise.

On completion of the counting of the keys, simply list the keys as follows

```python
for key in D.keys():
  print (key, D[key])
```