Fundamentals of Computer Science — programming exam, December 15th, Part 2

Instructions

1. You can write your solutions using either Spyder or Jupyter notebook.
2. Upload all your solutions in a single file to the by the end of the exam.
3. Make sure you upload all your solutions. If you are not sure, try again, or ask for help.

Additional notes

1. The exam is **not open book!**
2. For each problem, several input-output tests are provided. Make sure that your program works on those (and other) tests.
3. Try to write, **and debug** your code to the point it works.
4. If you did not manage to fully debug the code in time, submit it anyway. Solid attempts with some minor mistakes might still get a lot of partial points.
5. That being said, not all bugs will be treated equally. For example, try to make sure that your code doesn't have any "Syntax errors" when you are submitting it.

Problem 1 – Empirical Average

Write a function `randomized_average(f, n, sample_size)` that takes as input:

- a function `f`,
- a non-negative integer `n`,
- a positive integer `sample_size`.

You may assume that `f` accepts an integer input between `0` and `n` (inclusive) and returns a `float`.

Your task is to:

1. Generate `sample_size` random integers uniformly from the range `0` to `n` (inclusive).
2. Evaluate `f` on each of these randomly chosen inputs.
3. Return the average of the resulting values.

Examples

The output of `randomized_average` is **not deterministic**, since it depends on random sampling. Therefore, the function will not always return the exact same value. The examples below demonstrate how your function should be used and what the output should be *approximately*.

```
def f(x):
    return x / 2

result = randomized_average(f, 100, 10_000)
if abs(result - 25) < 0.3:
    print("OK")
```

```
def g(x):
    if x >= 50:
        return 1.
    else:
        return 0.

result = randomized_average(g, 99, 10_000)
if abs(result - 0.5) < 0.1:
    print("OK")`
```

Reminder

To generate a random integer between `a` and `b` (inclusive), first import the `random` module:

```
import random
```

Then use:

```
random.randint(a, b)
```

Problem 2 – Counting Connected Components

Write a function `count_connected_components(graph)` that takes a graph as input and returns the number of connected components in the graph.

The graph is represented as a list of length `n`, where each element is a list of integers. Specifically, `graph[i]` contains the neighbors of vertex `i`. You may assume that the graph is **undirected**: if vertex `j` appears in `graph[i]`, then vertex `i` also appears in `graph[j]`. The vertices are numbered from `0` to `n - 1` (inclusive).

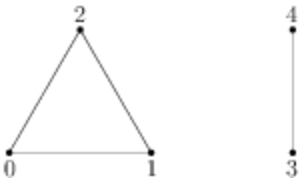
To receive full credit, your implementation should run in time $O(n+m)$ where `n` is the number of vertices and `m` is the number of edges.

Examples

Example 1

```
graph = [[1, 2], [0, 2], [0, 1], [4], [3]]
count_connected_components(graph) == 2
```

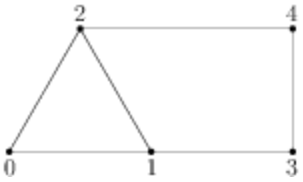
This input corresponds to the graph shown below, which has two connected components.



Example 2

```
graph = [[1, 2], [0, 2, 3], [0, 1, 4], [4, 1], [3, 2]]
count_connected_components(graph) == 1
```

This input corresponds to the graph shown below, which has a single connected component.



Example 3

```
graph = [[], [], [], [], []]
count_connected_components(graph) == 5
```

This input represents an empty graph with 5 vertices and no edges. In this case, each vertex forms its own connected component, so the correct output is 5.

Hint

A standard way to solve this problem is to use a variant of the Depth-First Search (DFS) algorithm discussed in the course.

Problem 3 – Local Permutations

A sequence σ of the numbers 0 to $n-1$ (inclusive) is called a k**-local permutation** if it satisfies the following two conditions:

1. **Permutation property:**
   Each number from 0 to $n-1$ appears exactly once in the sequence $\sigma(0), \sigma(1), \ldots, \sigma(n-1)$.
2. **Locality property:**
   For every index $i \in \{0, \ldots, n-1\}$,

   $$|\sigma(i) - i| \leq k.$$

   Your task is to write a function `count_local_permutations(n, k)` that takes integers `n` and `k` as input and returns the number of k-local permutations of the numbers 0 to $n-1$.

Examples

Example 1

```
count_local_permutations(10, 0) == 1
```

For any n, there is exactly one 0-local permutation:

$$(0, 1, \ldots n-1).$$

Example 2

```
count_local_permutations(4, 1) == 5
```

There are 5 permutations of $(0, 1, 2, 3)$ that are 1-local. They are:

```
[0, 1, 2, 3]
[0, 1, 3, 2]
[0, 2, 1, 3]
[1, 0, 2, 3]
[1, 0, 3, 2]
```

Example 3

```
count_local_permutations(4, 2) == 14
count_local_permutations(4, 3) == 24
count_local_permutations(10, 1) == 89
count_local_permutations(10, 2) == 2177
count_local_permutations(10, 3) == 19708
```

Hints

- A straightforward solution that generates all permutations of the numbers 0 to $n - 1$ and checks which ones are k-local runs in time $O(n!)$. Such a solution will receive a significant amount of partial credit.
- To receive full credit, your solution should run in time $O(k)^n$.
- One effective approach is to use a **backtracking algorithm**. For example, you may write a recursive function `backtracking(n, k, prefix)` that returns the number of ways the partial permutation `prefix` can be extended to a full k-local permutation. This function can work as follows:
  1. If the length of `prefix` is `n`, then `prefix` is a valid k-local permutation; return `1`.
  2. Otherwise, determine the range of valid values for the next element. Specifically, the next value must lie between
     ```
     max(0, len(prefix) - k)
     ```
     and
     ```
     min(n - 1, len(prefix) + k)
     ```
     (inclusive).
  3. For each integer `j` in this range that does not already appear in `prefix`, recursively call the backtracking function with `prefix + [j]`.

4. Sum the results of all recursive calls and return this total.