Problem 1: Generate a Random DNA Sequence

Description: Create a random DNA string with letters from the whole alphabet A, C, G, and T. First make a list of random letters and then join all those letters to a string. Also write another function to count the number of bases in the random sequence and measure the CPU time for large such DNA strings. (Hint : use import random, import time)

Reference

Illustrating Python via Bioinformatics Examples, Hans Petter Langtangen, Geir Kjetil Sandve, https://hplgit.github.io/bioinf-py/doc/pub/html/main_bioinf.html

```
import random
import time

def randomDNA(n):
  randomList = [random.choice(['A', 'C', 'G', 'T']) for i in range(n)]
  dna = ''.join(randomList)
  return dna

def BaseCount_CpuTime(dna):
  t1 = time.process_time() #start time
  count = {'A': 0, 'C': 0, 'G': 0, 'T': 0}
  for base in dna:
      count[base] += 1
  t2 = time.process_time() #end time
  cpuTime = t2-t1
  return count, cpuTime
```

```
n = int(input("Enter the length of dna sequence : "))
dna = randomDNA(n)
count, CPUtime = BaseCount_CpuTime(dna)

print(f"Random DNA sequence of length {n} :", dna)
print("Number of bases in the sequence : ", count)
print("CPU time to count the bases : ", CPUtime)
```

```
Enter the length of dna sequence : 10000
Random DNA sequence of length 10000 : TTATTGATACTAGGCACAGTTTTTTTTGTGTATGGAAATACTCTAAT
Number of bases in the sequence :  {'A': 2452, 'C': 2526, 'G': 2490, 'T': 2532}
CPU time to count the bases :  0.0010613749999990318
```

Problem 2: Compute the Hamming Distance Between Two Strings

We say that position i in k-mers $p_1 \ldots p_k$ and $q_1 \ldots q_k$ is a mismatch if $p_i \neq q_i$. For example, CGAAT and CGGAC have two mismatches. The number of mismatches between strings p and q is called the Hamming distance between these strings and is denoted $HammingDistance(p, q)$.

```
def HammingDistance(p, q):
  distance = 0
  for i in range(len(p)):
    if p[i] != q[i]:
      distance += 1
  return distance
```

```
HammingDistance("GGGCCGTTGGT", "GGACCGTTGAC")
```

```
3
```

Problem 3: Find Patterns Forming Clumps in a String

Given integers L and t, a string *Pattern* forms an (L, t)-clump inside a (larger) string *Genome* if there is an interval of *Genome* of length L in which *Pattern* appears at least t times.

For example, TGCA forms a (25,3)-clump in the following *Genome*: gatcagcataagggtcccTGCAATGCATGACAAGCCTGCAgttgttttac

```
def ClumpFinding(genome, k, L, t):
  clumps = set()

  for i in range(len(genome) - L + 1):
      window = genome[i:i+L]
      count = {}
      for j in range(L - k + 1):
          kmer = window[j:j+k]
          count[kmer] = count.get(kmer, 0) + 1

      for kmer, frequency in count.items():
          if frequency >= t and kmer not in clumps:
              clumps.add(kmer)

  return clumps
```

```
genome = "CGGACTCGACAGATGTGAAGAAATGTGAAGACTGAGTGAAGAGAAGAGGAAACACGACACGACATTGCGACATAATGTA
k,l,t = 5, 75, 4
ClumpFinding(genome, k, l, t)
```

Problem 4: Find a Position in a Genome Minimizing the Skew

Define the **skew** of a DNA string *Genome*, denoted *Skew*(*Genome*), as the difference between the total number of occurrences of 'G' and 'C' in *Genome*. Let $Prefix_i$ (*Genome*) denote the **prefix** (i.e., initial substring) of *Genome* of length *i*. For example, the values of *Skew*($Prefix_i$ ("CATGGGCATCGGCCATACGCCCATGGGCATCGGCCATACGCC")) are:

0 -1 -1 -1 0 1 2 1 1 1 0 1 2 1 0 0 0 0 -1 0 -1 -2

```
def skew(genome):
  count = {'G':0, 'C':0}
  skewlist=[]
  for i in genome:
    if i in count:
      count[i]+=1
    skewlist.append(count['G']-count['C'])
  return [i+1 for i, val in enumerate(skewlist) if val == min(skewlist)]
```

```
genome = "CCTATCGGTGGATTAGCATGTCCCTGTACGTTTCGCCGCGAACTAGTTCACACGGCTTGATGGCAAATGGTTTTTCCGGC
print(skew(genome))
```

```
[53, 97]
```