Universidad Andrés Bello

FACULTAD DE INGENIERIA

CARRERA DE INGENIERIA CIVIL INFORMATICA

CIENCIA ABIERTA:

RESOLUCIÓN DE EJERCICIOS DE BIOINFORMÁTICA DE ROSALIND

Autor:

Edgar Alejandro Ramos Vivenes

Profesor Tutor:

Dr. Gustavo Esteban Gatica

Santiago, Chile.

2024

Tabla de contenido

Python Village 4
Installing Python 4
Variables and Some Arithmetic 4
String and Lists 5
Conditions and Loops 5
Working with Files 6
Dictionaries 6
Algoritmitcs Heights 7
Fibonacci Numbers 7
Binary Search 8
Degree Array 9
Insertion Sort 10
Double Degree Array 11
Majority Element 11
Merge Two Sorted Arrays 12
2SUM 13
Breadth First Search 14
Connected Components 15
Building a Heap 15
Merge Sort 16
2-way Partition 17
3SUM 18
Testing Bipartiteness 19
Testing Acyclicity 19
Dijkstra's Algorithm 20
Heap Sort 21
Counting Inversions 21
3-way Partition 22
Square in a Graph 23
Bellman-Ford Algorithm 24
Shortest Cycle Through a Given Edge 24

Median 25
Partial Sort 26
Topological Sorting 26
Hamiltonian Path in DAG 26
Negative Weight Cycle 27
Quick Sort 28
Strongly Connected Components 29
2-Satisfiability 30
General Sink 30
Semi-Connected Graph 31
Shortest Paths in DAG 32
Bioinformatics Armory 34
Introduction to the Bioinformatics Armory 34
GenBank Introduction 34
Data Formats 35
FASTQ format introduction 36
Read Quality Distribution 36
Protein Translation 37
Read Filtration by Quality 38
Complementing a Strand of DNA 39
Base Quality Distribution 40
Finding Genes with ORFs 41
Base Filtration by Quality 42
Bioinformatics Stronghold 44
Counting DNA Nucleotides 44
Transcribing DNA into RNA 44
Complementing a Strand of DNA 45
Rabbits and Recurrence Relations 46
Computing GC Content 47
Counting Point Mutations 48
Mendel's First Law 49
Translating RNA into Protein 50
Finding a Motif in DNA 51

Consensus and Profile 52
Mortal Fibonacci Rabbits 53
Overlap Graphs 54
Calculating Expected Offspring 55
Finding a Share Motif 56
Independent Alleles 57
Finding a Protein Motif 58
Inferring a mRNA from Protein 59
Open Reading Frames 60
Enumerating Gene Orders 61
Calculating Protein Mass 62
Longest Increasing Subsequence 63
Enumerating Oriented Gene Orderings 64

Python Village

Installing Python

- Descripción:

Se resalta que la página Rosalind.info recomienda realizar los ejercicios en Python, ya que para los nuevos usuarios puedan aprender mediante un lenguaje simple, poderoso e incluso divertido.

- Resolución:

Este problema se encarga que los usuarios instalen el intérprete de Python respectivamente para utilizar su IDLE, y realicen una prueba para verificar si está en funcionamiento, ejecutando simplemente un "import this", lo cual resulta en la impresión de un poema por consola.

https://rosalind.info/problems/ini1/

Variables and Some Arithmetic

- Descripción:

En Python, existen diferentes tipos de datos. Por ejemplo, los más básicos son los enteros, los flotantes y las cadenas de caracteres. Enteros y flotantes se usan en operaciones aritméticas, como suma, resta, multiplicación, división, etc. Por otro lado, las cadenas de caracteres son tipos de datos específicos, ya que consisten en secuencias ordenadas de letras, números, símbolos y otros caracteres.

- Resolución:

Se tienen dos números enteros positivos menores que 1000, A y B. Dado esos datos de entrada, se pide devolver el valor de la hipotenusa al cuadrado tomando los valores de A y B como los catetos de un triángulo. Usando el teorema de Pitágoras, el cuadrado de la hipotenusa es equivalente a la suma de los cuadrados de sus catetos. Por lo que, se puede elevar tanto A y B al exponente 2 y sumar los resultados de ambos catetos. Existen dos formas para obtener el cuadrado de un número, la primera es usando los operadores nativos de Python para las potencias y, la segunda, es usando la función **pow (base, exponente)**.

https://rosalind.info/problems/ini2/

String and Lists

- Descripción:

Como se mencionó anteriormente, existen varios tipos de datos, como cadenas de caracteres, enteros y flotantes. No obstante, hay estructuras de datos diseñadas para almacenar estos diferentes tipos de datos de manera sencilla; un ejemplo excelente de esto son las listas. En la descripción de las listas, se explican los métodos de acceso a cualquier elemento o conjunto de elementos dentro de ellas. Además, se destaca que las cadenas de caracteres comparten funciones similares o comunes con las listas, ya que pueden considerarse como una lista de caracteres.

- Resolución:

Se espera como entrada una cadena de texto S con una longitud mínima de 200 caracteres, seguida de cuatro números enteros. A partir de estos datos, el objetivo es devolver dos segmentos de la cadena S, considerando cada par de números como un punto de inicio y fin para cada segmento. Una vez que se tienen los cuatro números, el resultado será una nueva cadena de texto que contenga ambos segmentos separados por un espacio. Este proceso se puede llevar a cabo utilizando métodos tanto de listas como de cadenas. Gracias a su comportamiento, es posible especificar los puntos de inicio y final de cada segmento dados los datos de la cadena de texto S.

https://rosalind.info/problems/ini3/

Conditions and Loops

- Descripción:

En Python, se utilizan las condicionales (if, elif, else) para elegir entre diferentes opciones. Estas estructuras permiten tomar decisiones basadas en valores específicos establecidos. Asimismo, Python ofrece funciones conocidas como bucles, que repiten un fragmento de código o funciones de manera continua hasta que se cumpla una condición determinada. Entre los bucles disponibles en Python se encuentran el bucle for y el bucle while.

- Resolución:

Se tienen dos números enteros positivos, A y B, que cumplen con la condición (A < B < 10000). Se requiere calcular la suma total de los números impares dentro del rango de A hasta B. Para resolver este problema, se optó por utilizar un bucle for junto con la función range(), la cual establece una lista de elementos cuyos valores van desde A hasta B. Posteriormente, se empleó la función sum() en conjunto con una condicional que verifica si el número en la iteración del bucle for no es divisible

entre 2, lo cual determina que es impar, dado que todo número par puede ser dividido por 2.

https://rosalind.info/problems/ini4/

Working with Files

- Descripción:

Dentro de este problema, se explora la diversidad de métodos disponibles para leer el contenido de un archivo utilizando la función open(). Teniendo en cuenta el manejo previo de listas y cadenas, un archivo .txt puede ser dividido por líneas, lo que resulta en cadenas de texto que son almacenadas en una lista de cadenas. Es importante destacar que los bucles desempeñan un papel fundamental tanto en la lectura como en la escritura de un archivo.

- Resolución:

Como ejercicio, consideremos un archivo de texto (archivo.txt) con un máximo de 1000 líneas. El contenido de este archivo es irrelevante para el propósito de este problema, ya sea un texto científico, un poema o la letra de una canción. Se solicita crear un nuevo archivo.txt similar al original, pero conteniendo únicamente el contenido de las líneas pares del archivo original.

Tratando el archivo.txt como una lista de cadenas de texto, es decir, una lista de las líneas de texto del archivo, podemos iterar a través de ellas utilizando un bucle, preferiblemente for. Al emplear la función enumerate() junto con la lista de cadenas, obtenemos un contador del elemento que se está iterando. Este contador va desde 0 hasta N-1 líneas del archivo.txt. Siguiendo un enfoque inverso al utilizado en el problema anterior, podemos ajustar la condición para verificar si el número de línea, determinado por el contador, es par o no. Luego, podemos agregar estas líneas pares a un nuevo archivo.

https://rosalind.info/problems/ini5/

Dictionaries

- Descripción:

Los diccionarios son otra estructura de datos fundamental en Python. Se distinguen por almacenar datos de manera similar a las listas, pero en lugar de utilizar índices de posición para acceder a los datos, emplean un enfoque de clave-valor. Esto significa que cada valor se guarda asociado a una clave única. Todos los datos mencionados se pueden guardar como valores en un diccionario: enteros, cadenas, listas e incluso otros diccionarios u objetos.

- Resolución:

Como dato de entrada, se recibe una cadena S con una longitud máxima de 10000 letras. El problema consiste en devolver la cantidad de apariciones de cada palabra en la cadena, precedida de dicha palabra.

Para abordar este problema, en primer lugar, se emplea la función split(), que devuelve una lista de subcadenas llamada WORDS obtenidas al dividir la cadena S por espacios, lo que resulta en una lista que contiene todas las palabras que conforman S.

A continuación, se crea un diccionario y se implementa un bucle for para iterar sobre cada palabra de la lista WORDS y contabilizar sus apariciones. Se suma 1 unidad al valor correspondiente en el diccionario utilizando las palabras como claves.

Finalmente, se utiliza otro bucle for para recorrer el diccionario y así obtener una lista que contenga todas las palabras y la cantidad de veces que aparecen en la cadena S. De esta manera, se completa la tarea de contar las apariciones de cada palabra en la cadena.

https://rosalind.info/problems/ini6/

Algoritmitcs Heights

Fibonacci Numbers

- Descripción:

Se destaca la rapidez de crecimiento de los números de Fibonacci, casi tan rápida como las potencias de 2. Por ejemplo, F30 es más de un millón, y F100 ya tiene 21 dígitos.

Ninguna otra secuencia de números ha sido estudiada tan extensamente ni aplicada a tantos campos, como la biología, la demografía, el arte, la arquitectura y la música, por mencionar algunos. Además, junto con las potencias de 2, es la secuencia favorita de la informática.

- Resolución:

El problema consiste en generar los números de Fibonacci, que se rigen por la regla Fn = Fn-1 + Fn-2, con F1 = 1 y F0 = 0. Se requiere calcular el valor de Fn dado un número entero positivo N ≤ 25.

Para ello, se implementó una función llamada fibonacci() que, utilizando programación dinámica, toma un parámetro N. Luego, inicializa una lista llamada fibo_dp con los primeros dos números de Fibonacci (0 y 1).

Despues, si N es menor o igual a 1, la función devuelve N. En caso contrario, se ejecuta un bucle para calcular los números de Fibonacci hasta llegar a N, almacenándolos en la lista fibo_dp.

Finalmente, la función devuelve el número de Fibonacci correspondiente a N.

https://rosalind.info/problems/fibo/

Binary Search

- Descripción:

Se destaca la eficacia del algoritmo de búsqueda binaria como el máximo exponente del enfoque de dividir y conquistar.

Para encontrar una clave k en un archivo grande que contiene claves A[1..n] ordenadas, primero comparamos k con A[n/2] y, según el resultado, recursamos en la primera mitad del archivo, A[1..n/2], o en la segunda mitad, A[n/2+1..n]. La recurrencia ahora es T(n) = T(n/2) + O(1). Al aplicar el teorema maestro (con a=1, b=2, d=0), obtenemos la solución conocida: un tiempo de ejecución de solo O(logn).

- Resolución:

El problema consiste en encontrar un conjunto dado de claves en un arreglo dado. Se nos proporcionan dos enteros positivos n ≤ 10^5 y m ≤ 10^5, un arreglo ordenado A[1..n] de enteros de -10^5 a 10^5, y una lista de m enteros -10^5 ≤ k1, k2,..., k_m ≤ 10^5.

Se solicita, para cada k_i, proporcionar un índice 1 ≤ j ≤ n tal que A[j] = k_i o "-1" si no existe dicho índice.

Para la solución del problema, se implementaron dos funciones relacionadas con la búsqueda de elementos en una lista ordenada, utilizando el algoritmo de búsqueda binaria.

La primera función llamada binary_search(), como su nombre en inglés lo indica, implementa el algoritmo de búsqueda binaria. Esta función toma dos parámetros: una lista ordenada (array) y un elemento a buscar (k). Comienza inicializando punteros start y end al principio y al final de la lista respectivamente. Luego, mientras el puntero start sea menor o igual que el puntero end, calcula el punto medio (middle) de la lista. Luego, compara el elemento medio con el elemento buscado. Si el elemento medio es menor que el elemento buscado, actualiza el puntero start para que apunte al siguiente elemento después del medio. Si el elemento medio es

mayor que el elemento buscado, actualiza el puntero `end` para que apunte al elemento anterior al medio. Si el elemento medio es igual al elemento buscado, devuelve la posición del elemento medio. Si el bucle termina sin encontrar el elemento, la función devuelve -1.

La segunda función, llamada searching_elements(), toma dos listas como entrada: ListOrder, que se espera que esté ordenada, y ListElement, que contiene los elementos que se buscan en ListOrder. Itera sobre los elementos en ListElement. Para cada elemento, llama a la función binary_search() para encontrar su posición en ListOrder. Luego, concatena las posiciones encontradas en una cadena de salida. Si un elemento no se encuentra en ListOrder, su posición se representa como -1 en la cadena de salida.

Finalmente, el resultado de llamar a la función searching_elements() se almacena en la variable OUTPUT, es decir, se almacena la solucion al problema.

https://rosalind.info/problems/bins/

Degree Array

- Descripción:

Se explica cómo los grafos pueden ayudarnos a resolver diversos problemas de manera clara y precisa. Por ejemplo, menciona el desafío de colorear un mapa político utilizando el menor número de colores posible, asegurándonos de que países vecinos tengan colores diferentes. Esto puede ser complicado debido a la complejidad del mapa, con sus fronteras intrincadas y otros detalles irrelevantes. Sin embargo, al representar el mapa como un grafo, donde cada país es un punto y las fronteras son líneas que conectan estos puntos, simplificamos el problema. En lugar de lidiar con todos los detalles del mapa, nos enfocamos solo en la información relevante para la coloración.

También menciona cómo este enfoque se aplica en otros contextos, como la programación de exámenes universitarios. Aquí, cada examen se representa como un punto en el grafo, y las restricciones de tiempo entre exámenes se representan como líneas que conectan estos puntos. De esta manera, podemos abordar el problema de asignar horarios a los exámenes de manera más estructurada y eficiente.

Ademas, se introduce la idea de grafos dirigidos, donde las relaciones entre los puntos tienen una dirección. Por ejemplo, en el caso de la World Wide Web, cada sitio web puede ser un punto en el grafo, y los enlaces entre ellos son líneas direccionales. Esto permite comprender la estructura de la web y su conectividad de manera más detallada.

- Resolución:

El problema consiste en analizar un grafo simple dado en formato de lista de aristas, con un máximo de n ≤ 103 vértices. En donde se requiere devolver un arreglo D[1..n] donde D[i] representa el grado del vértice i.

En un grafo no dirigido, el grado d(u) de un vértice u es el número de vecinos que tiene u, o equivalentemente, el número de aristas incidentes sobre él.

Para ello, se desarrollo la funcion counting_degree_vertex(), la cual toma un nodo y un diccionario como entrada. Si el nodo no está presente en el diccionario, lo agrega con un grado de 1. Si el nodo ya está en el diccionario, incrementa su grado en 1.

El proceso principal lee un archivo de entrada que contiene información sobre los vértices y las aristas de un grafo. Inicializa un diccionario vacío llamado DEGREE_VERTEXS. Luego, lee las líneas del archivo, extrayendo los nodos de cada arista y llamando a la función counting_degree_vertex() para contar el grado de cada nodo. Después de procesar todas las líneas del archivo, los vértices y sus grados se almacenan en el diccionario DEGREE_VERTEXS.

Finalmente, los vértices y sus grados se ordenan y se almacenan en la variable VERTEXS. Luego, se genera una cadena de salida concatenando los grados de los vértices ordenados y se almacena en la variable OUTPUT.

https://rosalind.info/problems/deg/

Insertion Sort

- Descripción:

Se describe la ordenación por inserción como un algoritmo de ordenación simple que construye el array final ordenado uno a la vez. Se señala que, aunque es menos eficiente en listas grandes en comparación con algoritmos avanzados como "Quick Sort", "Heap Sort" o "Merge Sort", ofrece ventajas como una implementación sencilla, eficiencia para conjuntos de datos pequeños y un espacio extra constante O(1).

Se destaca que, a pesar de tener un tiempo de peor caso cuadrático O(n^2), la ordenación por inserción se elige cuando los datos están casi ordenados o cuando el tamaño del problema es pequeño debido a su adaptabilidad y bajo sobrecoste. Se menciona que, por su estabilidad, a menudo se utiliza como caso base recursivo para algoritmos de ordenación de división y conquista con mayor sobrecoste, como "Merge Sort" o "Quick Sort".

- Resolución:

Se proporciona un número entero positivo n, no mayor a 10^3, y un array A[1..n] de enteros. El objetivo es determinar el número de intercambios realizados por el algoritmo de ordenación por inserción en el array dado.

Para la resolución del problema, se desarrolla la función insertion_sort() que toma dos parámetros: una lista (array) que contiene los elementos a ordenar y el número de elementos (n_elements) en la lista. La función itera a través de la lista, comenzando desde el segundo elemento hasta el último. Para cada elemento, compara su valor con los elementos anteriores y lo mueve hacia la izquierda hasta que esté en la posición correcta. Durante este proceso, realiza un seguimiento del número de intercambios realizados para mantener un registro de la cantidad de operaciones necesarias para ordenar la lista.

Una vez completada la ordenación, la función devuelve el número de intercambios realizados durante el proceso.

https://rosalind.info/problems/ins/

Double Degree Array

- Descripción:

Se centra en el problema de un grafo dado en formato de lista de aristas. Se especifica que se proporciona un grafo simple con un máximo de n vértices, donde n es menor o igual a 103.

- Resolución:

El objetivo es devolver un array D[1..n], donde D[i] es la suma de los grados de los vecinos del vértice i.

Se comienza leyendo un archivo de entrada que contiene información sobre los vértices y las aristas del grafo. Luego, utiliza la función counting_degree_vertex() para contar los grados de los vértices y sus vecinos, almacenando esta información en un diccionario llamado DEGREE_NEIGHBORS.

Finalmente, se genera una salida que muestra el grado total de los vecinos de cada vértice en el grafo. Si un vértice no tiene vecinos, se muestra un grado de 0. La salida se almacena en una cadena llamada OUTPUT, que contiene los grados de los vecinos de cada vértice, separados por espacios.

https://rosalind.info/problems/ddeg/

Majority Element

- Descripción:

Se aborda el concepto de un elemento mayoritario en un array A[1..n], definiéndolo como aquel cuyas entradas son más de la mitad del total de elementos.

Se especifica que se proporciona un número entero positivo k, no mayor a 20, y otro positivo n, no mayor a 10^4. Además, se entregan k arrays de tamaño n, los cuales contienen enteros positivos que no exceden 10^5.

- Resolución:

El objetivo es devolver, para cada array, un elemento que ocurra estrictamente más de n/2 veces si tal elemento existe, y "-1" en caso contrario.

Para ello, se implemento la funcion majority_element_search() que toma una lista de números (nums) y un parámetro (n) opcional que representa la longitud de la lista. Primero, busca un candidato a elemento mayoritario en la lista mediante un proceso de votación. Luego, verifica si el candidato realmente es el elemento mayoritario contando cuántas veces aparece en la lista.

Para encontrar el candidato a elemento mayoritario, el algoritmo recorre la lista de números y utiliza una técnica llamada algoritmo de Boyer-Moore. Este algoritmo mantiene un contador para el candidato actual y actualiza el contador cada vez que encuentra un nuevo número. Si el contador llega a cero, se elige un nuevo candidato. Cuando se encuentre un candidato, se cuenta cuántas veces aparece en la lista.

Finalmente, se genera una salida que indica si se encontró un elemento mayoritario en cada lista de números proporcionada. Si se encuentra un elemento mayoritario, se muestra su valor. Si no se encuentra, se muestra -1. La salida se almacena en una cadena llamada OUTPUT.

https://rosalind.info/problems/maj/

Merge Two Sorted Arrays

- Descripción:

Se menciona que, el procedimiento de fusión es una parte esencial del Merge Sort. Se proporciona un número entero positivo n, no mayor a 10^5, y un array ordenado A[1..n] de enteros que van desde -10^5 hasta 10^5, junto con un número entero positivo m, no mayor a 10^5, y un array ordenado B[1..m] de enteros que van desde -10^5 hasta 10^5.

- Resolución:

El objetivo es devolver un array ordenado C[1..n+m] que contenga todos los elementos de A y B.

Se implementa una función llamada merge_arrays() que toma dos listas como entrada y devuelve una nueva lista que combina y ordena los elementos de ambas listas.

En primer lugar, la función merge_arrays() recibe dos parámetros: array1 y array2, que son las dos listas que se deben combinar. Luego, utiliza el operador de concatenación “+” para combinar las dos listas en una sola lista. Después, utiliza la función sorted() para ordenar los elementos de la lista combinada en orden ascendente.

Finalmente, se almacena el resultado obtenido de la funcion y, posteriormente, se convierte en una cadena de salida llamada OUTPUT. Utilizando la función map() junto con str() se convierten los elementos de la lista combinada en cadenas de caracteres, y luego la función join() se utiliza para unir estos elementos en una única cadena separada por espacios.

https://rosalind.info/problems/mer/

2SUM

- Descripción:

Se aborda un escenario donde se proporciona un entero positivo k, no mayor a 20, y otro entero positivo n, no mayor a 10^4. Además, se entregan k arrays de tamaño n, que contienen enteros que varían entre -10^5 y 10^5.

El objetivo es generar, para cada array A[1..n], dos índices diferentes p y q, donde $1 \leq p < q \leq n$, tal que A[p] sea igual al negativo de A[q] si existen tales elementos, y devolver "-1" en caso contrario.

- Resolución:

Se implementa una función llamada two_sum2() que toma una lista de listas como entrada y devuelve una cadena que representa las posiciones de los pares de elementos cuya suma es igual a cero en cada lista interna.

La función two_sum2() utiliza una función auxiliar interna llamada find_pairs() que busca pares de elementos en una lista que suman cero. Para hacer esto, itera sobre cada elemento de la lista y verifica si el elemento opuesto (el negativo del valor actual) está presente en un diccionario de resultados. Si encuentra una coincidencia, devuelve las posiciones de los dos elementos que suman cero. Si no encuentra ningún par que sume cero, devuelve "-1".

Después de definir la función find_pairs(), la función two_sum2() itera sobre cada lista en la lista de listas de entrada (arrays). Para cada lista, llama a find_pairs y agrega el resultado a una cadena de salida (output). Al final, devuelve la cadena de

salida que contiene las posiciones de los pares de elementos que suman cero en cada lista interna, separados por saltos de línea.

https://rosalind.info/problems/2sum/

Breadth First Search

- Descripción:

Se centra en el problema de utilizar la búsqueda en anchura (BFS, por sus siglas en inglés) para calcular las distancias más cortas desde una fuente única en un grafo dirigido y no ponderado.

Se especifica que se proporciona un grafo dirigido simple con un máximo de n vértices, donde n es menor o igual a 10^3, en formato de lista de aristas.

El objetivo es devolver un array D[1..n], donde D[i] representa la longitud del camino más corto desde el vértice 1 hasta el vértice i (donde D[1]=0). Si el vértice i no es alcanzable desde 1, se establece D[i] en -1.

- Resolución:

Se implementó un algoritmo de búsqueda en anchura (BFS) en un grafo dado. La función breadth_first_search() que se desarrolló, toma tres parámetros: graph, que es un diccionario de lista de adyacencias que representa el grafo; vertexes, que es el número total de vértices en el grafo; y start, que es el vértice de inicio para la búsqueda en anchura.

Inicialmente, se crea una lista llamada distances que almacenará las distancias desde el vértice de inicio hasta cada uno de los otros vértices en el grafo. Todas las distancias se inicializan como -1, excepto la distancia desde el vértice de inicio a sí mismo, que se establece en 0. También se crea una cola (queue) que almacenará los vértices que se están visitando.

El algoritmo de la funcion comienza colocando el vértice de inicio en la cola. Luego, mientras la cola no esté vacía, se extrae un vértice de la cola y se exploran todos sus vecinos. Si la distancia al vecino aún no se ha calculado (es -1), se actualiza la distancia y se agrega el vecino a la cola para su posterior exploración.

Una vez que se han explorado todos los vértices accesibles desde el vértice de inicio, se devuelve una lista distances que contiene las distancias mínimas desde el vértice de inicio hasta cada uno de los otros vértices en el grafo.

https://rosalind.info/problems/bfs/

Connected Components

- Descripción:

Se aborda el problema de utilizar la búsqueda en profundidad (DFS, por sus siglas en inglés) para calcular el número de componentes conectados en un grafo no dirigido dado.

Se especifica que se proporciona un grafo simple con un máximo de n vértices, donde n es menor o igual a 10^3, en formato de lista de aristas.

El objetivo es devolver el número de componentes conectados en el grafo.

- Resolución:

Se desarrolló la función connected_components1() que toma un parámetro (graph), que es un diccionario de lista de adyacencia que representa el grafo no dirigido.

Dentro de la función connected_components1(), se define una función interna dfs() que realiza una búsqueda en profundidad (DFS) desde un vértice dado. Esta función marca los vértices visitados y explora recursivamente todos los vecinos no visitados de un vértice dado.

La función principal connected_components1() inicializa un conjunto (visited) para almacenar los vértices visitados y un contador (components) para contar el número de componentes conectados en el grafo. Luego, itera sobre todos los vértices del grafo y, para cada vértice no visitado, incrementa el contador de componentes en uno y realiza una búsqueda en profundidad desde ese vértice utilizando la función dfs().

Al final, la función devuelve el número total de componentes conectados en el grafo.

https://rosalind.info/problems/cc/

Building a Heap

- Descripción:

Se comenta el problema de construir un montículo binario a partir de un array dado. Un montículo binario es una estructura de datos basada en un árbol binario que se utiliza frecuentemente para implementar colas de prioridad. A su vez, los montículos binarios pueden implementarse fácilmente utilizando un array si el árbol subyacente es un árbol binario completo. Los nodos del árbol tienen un orden natural: fila por fila, comenzando desde la raíz y moviéndose de izquierda a derecha dentro de cada fila. Si hay n nodos, este orden especifica sus posiciones 1, 2, n dentro del array.

Moverse hacia arriba y hacia abajo en el árbol se simula fácilmente en el array, utilizando el hecho de que el nodo número j tiene padre [j/2] e hijos 2j y 2j+1.

Este problema requiere construir un montículo a partir del array dado. Se proporciona un número entero positivo n, no mayor a 10^5, y un array A[1...n] de enteros que van desde -10^5 hasta 10^5.

El resultado consiste en un array permutado A que cumple con la propiedad de montículo binario máximo: para cualquier 2 ≤ i ≤ n, A [i/2] ≥ A[i].

- Resolución:

Se implementa un algoritmo para construir un heap máximo a partir de una lista de números. Para ello, se crea la función building_a_max_heap() que toma dos parámetros opcionales: array, que es la lista de números a partir de la cual se construirá el heap máximo, y n, que es el tamaño de la lista.

El algoritmo comienza recorriendo la lista desde el último elemento hasta el segundo, lo que permite iniciar desde la primera posición de los nodos que tienen hijos. Para cada elemento en esta iteración, se calcula su padre y se compara con este último. Si el elemento es mayor que su padre, se intercambian. Luego, se procede a hacer ajustes descendiendo a través del heap para asegurar que se mantenga la propiedad del heap máximo en cada nivel.

El proceso de ajuste se realiza mediante un bucle que verifica si el nodo actual tiene hijos y si es mayor que alguno de ellos. Si encuentra un hijo mayor, intercambia el nodo actual con el hijo más grande. Este proceso continúa hasta que el nodo actual es mayor que ambos hijos o hasta que no tiene hijos.

https://rosalind.info/problems/hea/

Merge Sort

- Descripción:

El problema implica organizar una serie de números, lo cual se puede abordar eficientemente mediante la técnica de divide y vencerás. Esta estrategia implica dividir la lista original en dos partes iguales, luego ordenar cada mitad de manera recursiva y finalmente combinar las dos sublistas ordenadas en una lista única. Este enfoque se asemeja al proceso de "fusionar dos arrays ordenados" que se ha tratado previamente.

Se proporciona un número entero positivo n, no mayor a 10^5, y un array A[1..n] de enteros que van desde -10^5 hasta 10^5. Se requiere obtener un array A[1..n] ordenado.

- Resolución:

Se implementa el algoritmo de ordenamiento merge sort mediante la función merge_sort() que toma una lista de números como entrada y la ordena en orden ascendente.

El algoritmo utiliza un enfoque de dividir y conquistar. Primero, divide la lista en dos mitades de tamaño similar. Luego, recursivamente ordena cada mitad por separado utilizando el mismo algoritmo. Una vez ordenadas ambas mitades, combina las mitades para obtener la lista final ordenada.

El proceso de combinación se realiza mediante un algoritmo de mezcla, donde compara los elementos de las dos mitades y los fusiona en orden ascendente en una lista auxiliar. Luego, actualiza la lista original con los elementos ordenados.

https://rosalind.info/problems/ms/

2-way Partition

- Descripción:

Se describe el problema de la partición de un array, una parte fundamental del algoritmo Quick Sort. Su principal objetivo es colocar el primer elemento de un array dado en su posición adecuada en un array ordenado. Esta partición puede ser implementada en tiempo lineal, mediante un solo escaneo del array dado.

Se proporciona un número entero positivo n, no mayor a 10^5, y un array A[1..n] de enteros que van desde -10^5 hasta 10^5.

El problema requiere devolver un array permutado B[1..n], que es una permutación de A, y existe un índice q, donde 1 ≤ q ≤n, tal que B[i] ≤ A[1] para todos los valores de i desde 1 hasta q-1, B[q] = A[1], y B[i] > A[1] para todos los valores de i desde q+1 hasta n.

- Resolución:

Para la solución del problema, se creó una función llamada two_way_partition() que toma dos parámetros opcionales: a, que es la lista de números que se va a particionar, y n, que es la longitud de la lista.

La función comienza creando una copia de la lista original para preservar los datos originales. Luego, selecciona un pivote (lamda) que es el primer elemento de la lista. El algoritmo de la función utiliza dos índices, p y q, que representan los límites izquierdo y derecho de la partición respectivamente.

Seguidamente, recorre la lista desde el índice p hasta q. Si el elemento actual es mayor que el pivote, se intercambia con el elemento en q, disminuyendo q en uno.

Si el elemento es menor o igual que el pivote, se deja en su lugar y se incrementa p en uno.

Cuando p y q se cruzan, todos los elementos menores que el pivote están a la izquierda de la lista y los elementos mayores están a la derecha. En este punto, se coloca el pivote en su posición final en la lista, garantizando que todos los elementos a su izquierda sean menores o iguales que él, y todos los elementos a su derecha sean mayores que él.

Finalmente, se devuelve la lista modificada con la partición realizada.

https://rosalind.info/problems/par/

3SUM

- Descripción:

El problema consiste en encontrar tres índices diferentes en un array A[1..n] donde se cumple la condición A[p]+A[q]+A[r]=0. Se especifica que se proporciona un número entero positivo k, no mayor a 20, y otro entero positivo n, no mayor a 10^4. Además, se entregan k arrays de tamaño n, que contienen enteros en el rango de -10^5 a 10^5.

El objetivo es generar, para cada array, tres índices distintos p, q y r, donde $1 \leq p < q < r \leq n$, tales que la suma de los elementos en esos índices sea igual a cero. Si no existe la combinación, se devuelve "-1".

- Resolución:

Para solucionar el problema, se implemento un algoritmo que busca tres índices en una matriz de números, de tal manera que la suma de los elementos en estos índices sea igual a cero. La función three_indices() toma tres parámetros: arrays, que es una matriz de números; k, que representa el número de matrices en la lista; y n, que es la longitud de cada matriz.

El algoritmo itera sobre cada matriz en la lista. Para cada matriz, crea un diccionario llamado numbers que asigna cada número en la matriz a su índice correspondiente. Luego, utiliza dos bucles anidados para seleccionar dos números diferentes en la matriz (denominados first y second), y busca si el negativo de su suma está presente en el diccionario numbers. Si se encuentra, se determina el índice correspondiente a ese número (denominado C) y se devuelve una cadena que indica los tres índices encontrados.

Si no se encuentra ningún conjunto de índices cuya suma sea cero, se devuelve "-1" para indicar que no se ha encontrado ninguna solución.

https://rosalind.info/problems/3sum/

Testing Bipartiteness

- Descripción:

El problema consiste en determinar si un grafo es bipartito o no. Un grafo bipartito es un grafo G = (V, E) cuyos vértices pueden ser divididos en dos conjuntos (V=V1∪V2 y V1∩V2=∅) de tal manera que no existan aristas entre vértices en el mismo conjunto (por ejemplo, si u,v ∈ V1, entonces no hay una arista entre u y v).

Existen varias maneras de formular esta propiedad. Por ejemplo, un grafo no dirigido es bipartito si y solo si puede ser coloreado con solo dos colores. Otra formulación: un grafo no dirigido es bipartito si y solo si no contiene ciclos de longitud impar.

Se proporciona un número entero positivo k, no mayor a 20, y k grafos simples en formato de lista de aristas, cada uno con un máximo de 10^3 vértices.

Se requiere imprimir "1" para cada grafo si es bipartito y "-1" en caso contrario.

- Resolución:

Se implementa un algoritmo para probar la bipartición de un grafo. Se desarrolla la función testing_bipartiteness1() que toma dos parámetros: graphs, que es una lista de diccionarios que representan los grafos a probar, y k, que es el número de grafos en la lista.

Dentro de la función testing_bipartiteness1(), se define una función interna llamada bip_test(). Esta función utiliza un enfoque de búsqueda en profundidad (DFS) para recorrer el grafo desde un vértice dado y asignar colores a los vértices. Durante el recorrido, comprueba si existe un ciclo impar en el grafo, lo que indicaría que el grafo no es bipartito.

Luego, la función principal testing_bipartiteness1() itera sobre cada grafo en la lista proporcionada. Para cada grafo, inicializa listas de visitados y colores, y llama a la función bip_test() para probar la bipartición del grafo desde el primer vértice. Si el grafo es bipartito, se agrega el índice del grafo a la cadena de salida; de lo contrario, se agrega "-1".

Finalmente, la función devuelve una cadena con los índices de los grafos bipartitos y "-1" para los que no lo son.

https://rosalind.info/problems/bip/

Testing Acyclicity

- Descripción:

Se presenta un problema donde se requiere analizar un conjunto de k (un entero positivo igual o menor a 20) grafos dirigidos simples en formato de lista de aristas.

Cada grafo tiene como máximo 10^3 vértices y 3*10^3 aristas. Se requiere determinar, para cada grafo, si es acíclico o no, indicando "1" en caso afirmativo y "-1" en caso contrario.

- Resolución:

Se desarrolla una función llamada testing_acyclicity(), la cual toma una lista de grafos como argumento, con la variable GRAPHS como conjunto de k grafos dirigidos simples dados. Esta función realiza un análisis de los grafos para determinar si son acíclicos o no.

Internamente, la función define dos subfunciones: dfs() y is_cyclical(). La función dfs (Depth-First Search) implementa un algoritmo de búsqueda en profundidad para verificar si hay ciclos en el grafo. La función is_cyclical() utiliza dfs() para recorrer todos los nodos del grafo y determinar si existe algún ciclo.

La función principal testing_acyclicity() itera sobre cada grafo en la lista proporcionada, y para cada uno de ellos llama a is_cyclical() para determinar su naturaleza acíclica o cíclica. Luego, acumula los resultados en una cadena de texto donde cada resultado está separado por un espacio y la devuelve como salida.

https://rosalind.info/problems/dag/

Dijkstra's Algorithm

- Descripción:

El problema consiste en utilizar el algoritmo de Dijkstra para calcular las distancias más cortas desde una única fuente dentro de un grafo dirigido que contiene pesos de arista positivos.

Se proporciona un grafo dirigido simple con pesos de arista positivos, que van del 1 al 10^3, y un máximo de n≤10^3 vértices en formato de lista de aristas.

Se espera obtener un arreglo D[1..n], donde D[i] representa la longitud del camino más corto desde el vértice 1 hasta el vértice i respectivo (D[1]=0). Si el vértice i no es alcanzable desde el vértice 1, se establece D[i] en −1.

- Resolución:

Se implementa el algoritmo de Dijkstra para encontrar las distancias más cortas desde un nodo origen dado en un grafo ponderado dirigido.

Se comienza inicializando un diccionario de distancias, donde cada nodo tiene una distancia inicial infinita, excepto el nodo de origen que tiene una distancia de 0. Luego, utiliza un bucle while para explorar los nodos en orden de distancia mínima utilizando una cola de prioridad (heap).

En cada iteración del bucle, selecciona el nodo con la distancia mínima no visitada y actualiza las distancias de sus vecinos si encuentra un camino más corto. El algoritmo finalmente genera una cadena de salida que representa las distancias más cortas desde el nodo origen hasta todos los demás nodos en el grafo. Si un nodo no es alcanzable desde el nodo origen, su distancia se establece en -1 en la salida.

https://rosalind.info/problems/dij/

Heap Sort

- Descripción:

Se describe el algoritmo de ordenación por montículos, el cual inicia transformando un arreglo dado en un montículo máximo. Posteriormente, se repiten dos pasos simples aproximadamente n-1 veces: primero, se intercambia el último elemento del montículo con su raíz y se reduce el tamaño del montículo en 1; segundo, se desplaza hacia abajo el nuevo elemento raíz a su posición adecuada.

El problema se define con un entero positivo n no mayor a 10^5 y un arreglo de enteros A[1..n] que van desde -10^^5 hasta 105. El problema requiere devolver un arreglo A ordenado.

- Resolución:

Se implementa el algoritmo de ordenación por montículos (Heap Sort) con la funcion llamada heap_sort() que toma un arreglo y su longitud como entrada, y utiliza una función interna heapify() para convertir el arreglo en un montículo máximo. Luego, intercambia repetidamente el primer elemento del montículo (que es el más grande) con el último elemento no ordenado del arreglo y reorganiza el montículo para mantener su propiedad de máximo. Este proceso de "hundir" el elemento más grande hasta su posición correcta y ajustar el montículo se repite hasta que todos los elementos estén ordenados. En consecuencia, el resultado es un arreglo ordenado en orden ascendente.

https://rosalind.info/problems/hs/

Counting Inversions

- Descripción:

Se describe la noción de inversión en un arreglo A[1..n], la cual se define como un par de índices (i,j) donde 1≤i<j≤n y A[i]>A[j]. Estas inversiones reflejan cuán lejos está el arreglo de estar ordenado: si ya está ordenado, no hay inversiones, mientras que si está ordenado en orden inverso, el número de inversiones es máximo.

Se proporcionan: un entero positivo n≤10^5 y un arreglo A[1..n] de enteros que varían entre -10^5 y 10^5. El problema requiere calcular el número de inversiones en A.

- Resolución:

Para la solución del problema, se desarrolló una función llamada counting_inversions1(), diseñada para calcular el número de inversiones en un arreglo dado. La función utiliza una estrategia de división y conquista para lograr esto.

El procedimiento principal de la función implica dividir recursivamente el arreglo en mitades, hasta que los subarreglos alcanzan una longitud de 1 o menos. Luego, se fusionan los subarreglos ordenados mientras se cuentan las inversiones encontradas durante la fusión.

El proceso de fusión consiste en comparar elementos de los subarreglos y colocarlos en orden ascendente en un nuevo arreglo. Durante este proceso, se cuentan las inversiones: cada vez que un elemento de la mitad derecha es menor que un elemento de la mitad izquierda, se incrementa el contador de inversiones en una cantidad igual al número de elementos restantes en la mitad izquierda.

Finalmente, el número total de inversiones se devuelve junto con el arreglo ordenado.

https://rosalind.info/problems/inv/

3-way Partition

- Descripción:

El problema planteado es similar a "Partición de 2 Vías", pero con la meta de particionar un arreglo de entrada de manera más precisa. Se proporciona un entero positivo n (menor o igual a 10^5) y un arreglo A[1..n] de enteros entre -10^5 y 10^5. Se debe retornar un arreglo B[1..n] que sea una permutación de A, donde existen índices 1 ≤ q ≤ r ≤ n tal que B[i] es menor que A[1] para 1 ≤ i ≤ q - 1, B[i] es igual a A[1] para q ≤ i ≤ r, y B[i] es mayor que A[1] para r + 1 ≤ i ≤ n.

- Resolución:

Se desarrolla una función llamada three_way_partition(), que realiza una partición de tres vías en un arreglo dado. La partición se realiza en torno a un pivote seleccionado como el primer elemento del arreglo. Luego, se mueven los elementos del arreglo para que los elementos menores que el pivote estén a la izquierda, los iguales al pivote estén en el medio y los mayores que el pivote estén a la derecha. Esto se logra utilizando cuatro punteros (leftLess, left, right, rightGreater) y dos bucles while que recorren el arreglo desde los extremos hacia el centro, intercambiando los elementos según sea necesario para lograr la partición de tres vías.

https://rosalind.info/problems/par3/

Square in a Graph

- Descripción:

El problema consiste en analizar una serie de grafos no dirigidos simples, cada uno representado en un formato de lista de aristas, donde se proporciona un número entero positivo k (donde k ≤ 20). Cada grafo tiene un máximo de n vértices (donde n ≤ 400).

El problema consiste en determinar, para cada grafo, si contiene un ciclo simple de longitud 4. En caso de existir dicho ciclo, la salida debe ser "1"; de lo contrario, "-1".

- Resolución:

Se implementa una función llamada count_cycles() que toma una lista de grafos y una longitud de ciclo como entrada. Internamente, utiliza una función auxiliar llamada DFS() que implementa el algoritmo de búsqueda en profundidad (DFS) para contar los ciclos simples de la longitud especificada en cada grafo.

El proceso comienza inicializando una lista de marcadores para realizar un seguimiento de los vértices visitados. Luego, itera a través de cada vértice en el grafo y aplica DFS para buscar ciclos de la longitud especificada. Una vez completada la búsqueda, se realiza un cálculo para determinar si se encontraron ciclos y se devuelve un valor correspondiente (1 si se encontraron ciclos, -1 de lo contrario).

Luego, se itera a través de una lista de grafos (GRAPHS) y se aplica la función count_cycles() a cada uno de ellos, acumulando los resultados en una cadena de salida llamada OUTPUT.

https://rosalind.info/problems/sq/

Bellman-Ford Algorithm

- Descripción:

Se describe la tarea de utilizar el algoritmo de Bellman-Ford para calcular las distancias más cortas desde una única fuente en un grafo dirigido con posibles pesos de arista negativos (pero sin ciclos negativos).

Se proporciona un grafo dirigido simple con pesos de arista enteros que van desde -10^3 hasta 10^3, y con un máximo de n≤10^3 vértices en formato de lista de aristas.

El problema consiste en devolver un arreglo D[1..n], donde D[i] representa la longitud de la ruta más corta desde el vértice 1 hasta el vértice i (D[1]=0). Si i no es alcanzable desde 1, se establece D[i] como x.

- Resolución:

Se implementa el algoritmo de Bellman-Ford para calcular las distancias más cortas desde un vértice de origen en un grafo dirigido con pesos de aristas, permitiendo pesos negativos (pero sin ciclos negativos). Se espera que el grafo se represente como un diccionario de listas de adyacencia, donde las claves son los nodos y los valores son listas de tuplas que contienen los vecinos y los pesos de las aristas.

El algoritmo inicializa un diccionario de distancias, donde cada nodo tiene asignada una distancia inicial infinita excepto el nodo de origen, que tiene una distancia de 0. Luego, itera sobre el grafo varias veces, actualizando las distancias de los nodos vecinos si se encuentra un camino más corto. Finalmente, comprueba si hay ciclos negativos y marca los nodos inalcanzables.

El resultado se devuelve como una cadena de texto que representa las distancias más cortas desde el nodo de origen a cada nodo del grafo, donde "x" indica que el nodo no es alcanzable y un número representa la distancia desde el nodo de origen hasta el nodo correspondiente.

https://rosalind.info/problems/bf/

Shortest Cycle Through a Given Edge

- Descripción:

Se requiere analizar k grafos dirigidos simples, cada uno con pesos de aristas enteros positivos y con un máximo de 10^3 vértices en formato de lista de aristas.

El objetivo es determinar, para cada grafo, la longitud del ciclo más corto que incluya la primera arista especificada, si existe dicho ciclo. Si no se encuentra el ciclo con la arista especificada, la salida deberá ser "-1".

- Resolución:

Primero, define una función de búsqueda en profundidad (DFS) para visitar todos los nodos alcanzables desde un nodo inicial dado. Luego, utiliza esta función para encontrar todos los nodos alcanzables desde el nodo final de la arista especificada.

Después, calcula las distancias más cortas desde el nodo final de la arista a todos los nodos alcanzables utilizando un algoritmo similar a Dijkstra. Finalmente, retorna la longitud del ciclo más corto que pasa por la arista especificada, si existe, o -1 en caso contrario.

El algoritmo se ejecuta en una lista de grafos proporcionados en el objeto GRAPHS, y genera una cadena de salida que contiene las longitudes de los ciclos más cortos para cada grafo en la lista.

https://rosalind.info/problems/cte/

Median

- Descripción:

El problema consiste en implementar un algoritmo aleatorizado de tiempo lineal para el problema de selección.

Se proporciona un entero positivo n ≤ 10^5 y una matriz A[1..n] de enteros que van desde -10^5 hasta 10^5, junto con un número positivo k ≤ n. Se requiere encontrar el k-ésimo elemento más pequeño de A.

- Resolución:

Se realiza la mplementación de la función kth_smallest_element1(), que utiliza un algoritmo aleatorizado para encontrar el k-ésimo elemento más pequeño en una matriz dada. La función utiliza recursión para dividir la matriz en subconjuntos más pequeños y seleccionar aleatoriamente un pivote para dividir la matriz en cada paso.

Dentro de la función principal kth_smallest_element1(), se definen dos funciones auxiliares: partition() y random_select(). La función partition() toma un pivote aleatorio y reorganiza los elementos de la matriz de manera que los elementos más pequeños que el pivote estén a la izquierda y los elementos más grandes estén a la derecha. La función random_select() utiliza la partición para seleccionar de manera recursiva el k-ésimo elemento más pequeño de la matriz.

Finalmente, la función principal kth_smallest_element1() llama a random_select() con los parámetros apropiados para iniciar el proceso de selección del k-ésimo elemento más pequeño de la matriz dada.

https://rosalind.info/problems/med/

Partial Sort

- Descripción:

El problema consiste en manejar una situación donde se proporciona un entero positivo n que no excede 10^5 y un conjunto de números A[1..n] que varían entre -10^5 y 10^5, junto con otro entero positivo k que no supera 1000.

El problema consiste en devolver los k elementos más pequeños de un arreglo ordenado A.

- Resolución:

Se implementa el algoritmo de ordenamiento merge sort para ordenar el arreglo A dado como entrada como se explicó anteriormente. Se utilizo para ordenar la lista y, posteriormente, tomar los primeros K elementos del arreglo.

Finalmente, se convierten en una cadena separada por espacios, que se guarda en la variable de salida OUTPUT.

https://rosalind.info/problems/ps/

Topological Sorting

- Descripción:

Se presenta un grafo dirigido acíclico simple con un máximo de n≤103 vértices en el formato de lista de aristas. Se solicita un ordenamiento topológico (una permutación de vértices) del grafo.

- Resolución:

Se crea una función llamada Topological_Sorting() que toma un grafo representado como un diccionario de lista de adyacencias y un conjunto de nodos. Esta función realiza un ordenamiento topológico del grafo, eliminando los nodos con grado de entrada cero en un bucle while, y agregando estos nodos al resultado. Finalmente, devuelve una cadena que representa el ordenamiento topológico de los nodos del grafo.

https://rosalind.info/problems/ts/

Hamiltonian Path in DAG

- Descripción:

El problema consiste en determinar la presencia de un camino hamiltoniano dentro de un grafo dado. Un camino hamiltoniano se define como un recorrido en un grafo

que visita cada vértice exactamente una vez. Si bien verificar la existencia de un camino hamiltoniano en un grafo se sabe que es una tarea desafiante, se vuelve relativamente sencillo cuando se trata de Grafos Acíclicos Dirigidos (DAGs).

La entrada para este problema consta de un entero positivo k, donde k ≤ 20, junto con k grafos acíclicos dirigidos simples representados en el formato de lista de aristas. Estos grafos contienen como máximo 10^3 vértices cada uno.

El objetivo del problema es generar una salida para cada grafo. Si un grafo contiene un camino hamiltoniano, la salida debe consistir en "1" seguido de la lista de vértices que constituyen el camino hamiltoniano. Por el contrario, si el grafo no contiene un camino hamiltoniano, la salida debe ser "-1".

- Resolución:

Para ello, se implementó una función llamada hamiltonian_path(), que toma tres parámetros: G, V y k. Esta función busca determinar si los grafos proporcionados contienen caminos hamiltonianos. Internamente, utiliza varias funciones auxiliares para realizar un ordenamiento topológico en un grafo dirigido acíclico (DAG) y verificar la validez de los caminos obtenidos. Luego, itera sobre cada grafo proporcionado, realiza el ordenamiento topológico y verifica si el camino resultante es consecutivo según las conexiones del grafo. Finalmente, genera un resultado indicando si el grafo contiene un camino hamiltoniano y devuelve una cadena que resume los resultados para cada grafo en la lista de entrada.

https://rosalind.info/problems/hdag/

Negative Weight Cycle

- Descripción:

El objetivo consiste en utilizar el algoritmo de Bellman-Ford para verificar si un grafo dado contiene un ciclo de peso negativo.

Se proporciona un entero positivo k (donde k ≤ 20) y k grafos dirigidos simples con pesos de arista en el rango de -10^3 a 10^3. Cada grafo tiene como máximo n (donde n ≤ 103) vértices y se presenta en formato de lista de aristas.

Para cada uno de los k grafos, se requiere devolver "1" si contiene un ciclo de peso negativo, y "-1" si no lo contiene.

- Resolución:

Se implementa el algoritmo de Bellman-Ford para encontrar el camino más corto desde un nodo fuente en un grafo ponderado.

El algoritmo comienza iterando sobre todos los nodos del grafo. Para cada nodo, inicializa un diccionario de distancias estableciendo todas las distancias a infinito, excepto la distancia al nodo fuente, que se establece en 0. Luego, realiza un bucle para iterar sobre todos los nodos del grafo varias veces, actualizando las distancias si se encuentra un camino más corto. El algoritmo repite este proceso hasta que no haya más cambios en las distancias o hasta que se haya realizado un número máximo de iteraciones.

Después de ejecutar el algoritmo, comprueba si hay ciclos de peso negativo en el grafo. Si se encuentra un ciclo de peso negativo, devuelve 1 para indicar que existe un ciclo de peso negativo en el grafo. De lo contrario, devuelve -1 para indicar que no hay ciclos de peso negativo.

https://rosalind.info/problems/nwc/

Quick Sort

- Descripción:

Al comparar los algoritmos de ordenamiento y el algoritmo de "Mediana", se observa que, más allá de la filosofía y estructura común de dividir y conquistar, son exactamente opuestos. "Merge Sort" divide el arreglo en dos de la manera más conveniente (primera mitad, segunda mitad), sin tener en cuenta las magnitudes de los elementos en cada mitad; pero luego trabaja arduamente para juntar los subarreglos ordenados. En contraste, el algoritmo de la mediana es cuidadoso con su división (primero los números más pequeños, luego los más grandes), pero su trabajo termina con la llamada recursiva.

Quick Sort es un algoritmo de ordenamiento que divide el arreglo exactamente de la misma manera que el algoritmo de la mediana; y una vez que los subarreglos están ordenados, mediante dos llamadas recursivas, no hay nada más que hacer. Su rendimiento en el peor de los casos es Θ(n^2), al igual que el de encontrar la mediana. Pero se puede demostrar que su caso promedio es O(nlogn); además, empíricamente supera a otros algoritmos de ordenamiento. Esto ha convertido a quicksort en un favorito en muchas aplicaciones, por ejemplo, es la base del código mediante el cual se ordenan archivos realmente enormes.

- Resolución:

Se implementa el algoritmo de ordenamiento QuickSort. Este algoritmo utiliza un enfoque de divide y vencerás para ordenar una matriz o lista de elementos. La función partition() se encarga de seleccionar un pivote (generalmente el último elemento del rango) y reorganizar los elementos de la matriz de manera que los elementos menores que el pivote estén a su izquierda, y los elementos mayores

estén a su derecha. Luego, la función devuelve la posición final del pivote después de la reorganización.

La función quickSort() utiliza la función partition() para dividir la matriz en subgrupos y luego aplica recursivamente el mismo proceso a cada subgrupo. Este proceso continúa hasta que todos los subgrupos tengan un solo elemento, lo que garantiza que toda la matriz esté ordenada cuando se complete la recursión.

En resumen, el algoritmo QuickSort para ordenar una matriz o lista de elementos en orden ascendente.

https://rosalind.info/problems/qs/

Strongly Connected Components

- Descripción:

El problema consiste en determinar el número de componentes fuertemente conectados en un grafo dirigido simple, representado en un formato de lista de aristas, donde el número de vértices es menor o igual a 10^3.

- Resolución:

Se define una clase llamada GFG que contiene tres métodos: dfs, isPath y findSCC.

El método dfs implementa una búsqueda en profundidad (DFS) recursiva para encontrar un camino desde un nodo de origen hasta un nodo de destino en un grafo representado por una lista de adyacencia. Retorna True si encuentra un camino, False de lo contrario.

El método isPath verifica si hay un camino entre dos nodos en el grafo utilizando el método dfs.

El método findSCC encuentra los componentes fuertemente conectados en un grafo representado por una lista de adyacencia. Retorna una lista de listas donde cada lista interna representa un componente fuertemente conectado.

Después de crear un objeto de la clase GFG, se llama al método findSCC con los parámetros V y EDGES, donde V es el número de nodos y EDGES es una lista de aristas.

El resultado se almacena en la variable ans, por lo que, se cuenta la función len(ans) para devolver el número de componentes.

https://rosalind.info/problems/scc/

2-Satisfiability

- Descripción:

El problema 2SAT implica asignar valores verdaderos o falsos a variables booleanas para satisfacer un conjunto de cláusulas donde cada cláusula es una disyunción de dos literales. Se busca encontrar una asignación que cumpla todas las cláusulas. Este problema se puede resolver eficientemente reduciéndolo al problema de encontrar componentes fuertemente conectados en un grafo dirigido.

Para ello, se construye un grafo donde cada variable y su negación son nodos, y las cláusulas se representan como aristas entre las negaciones de los literales. Si un componente fuertemente conectado contiene tanto una variable como su negación, la instancia no tiene una asignación satisfactoria. Por el contrario, si ningún componente contiene un literal y su negación, entonces la instancia es satisfactoria.

Este enfoque permite resolver el problema en tiempo lineal, lo que lo hace eficiente incluso para instancias grandes.

- Resolución:

Se implementa un algoritmo para resolver el problema 2SAT. Incluye funciones para realizar recorridos en profundidad (DFS) iterativos sobre el grafo dado, así como la función principal solve_2sat() que resuelve el problema 2SAT para un conjunto de cláusulas dado.

El bloque de código principal lee un archivo de entrada que contiene el número de casos k seguido de cláusulas de satisfacción de 2 literales para cada caso. Luego, construye grafos basados en estas cláusulas y resuelve el problema 2SAT para cada conjunto de cláusulas.

La salida final contiene el resultado de cada caso de 2SAT, indicando si es satisfactible y, en caso afirmativo, proporcionando una asignación satisfactoria de variables.

https://rosalind.info/problems/2sat/

General Sink

- Descripción:

Se enfrenta un problema que implica k (donde k es un entero positivo menor o igual a 20) grafos dirigidos simples. Cada grafo tiene como máximo 10^3 vértices y 2*10^3 aristas, representados en formato de lista de aristas.

El problema consiste en devolver, para cada grafo, un vértice desde el que todos los demás vértices son alcanzables (si existe tal vértice) y devolver "-1" sino existe.

- Resolución:

Se implementa una función llamada find_reachable_vertex() que toma un grafo representado como un diccionario de lista de adyacencia. Esta función busca un vértice desde el cual todos los demás vértices del grafo sean alcanzables. Utiliza un enfoque de búsqueda en profundidad (DFS) para explorar el grafo, comenzando desde cada vértice. Si encuentra un vértice desde el cual todos los demás son alcanzables, devuelve ese vértice. Si no encuentra tal vértice, devuelve -1.

https://rosalind.info/problems/gs/

Semi-Connected Graph

- Descripción:

El problema consiste en determinar si un grafo dirigido es semi-conectado. Un grafo es semi-conectado si, para todos los pares de vértices i, j, existe un camino desde i hasta j o desde j hasta i.

Se proporcionan k (donde k es un entero positivo menor o igual a 20) grafos dirigidos simples, cada uno con un máximo de 103 vértices, en formato de lista de aristas.

La tarea consiste en devolver, para cada grafo, el valor "1" si es semi-conectado y "-1" en caso contrario.

- Resolución:

Se implemento dos funciones principales y un bucle para procesar múltiples grafos. La primera función, BFS(), realiza un recorrido en anchura desde un vértice de inicio dado en un grafo dado. Esta función devuelve el orden de visita de los vértices y las distancias desde el vértice de inicio a cada vértice en el grafo.

La segunda función, ifSemiConnectedGraph(), verifica si un grafo dado es semi-conectado o no. Utiliza la función BFS() para obtener la distancia entre cada par de vértices en el grafo. Luego, construye una matriz de adyacencia y verifica si hay algún vértice que no esté conectado a todos los demás. Si encuentra algún vértice desconectado, devuelve -1, indicando que el grafo no es semi-conectado. Si no encuentra ningún vértice desconectado, devuelve 1, indicando que el grafo es semi-conectado.

Finalmente, el bucle principal procesa una lista de grafos llamada GRAPHS utilizando la función ifSemiConnectedGraph() y genera una cadena de salida que contiene "1" o "-1" para cada grafo, indicando si es semi-conectado o no, respectivamente.

https://rosalind.info/problems/sc/

Shortest Paths in DAG

- Descripción:

Existen dos subclases de grafos que automáticamente excluyen la posibilidad de ciclos negativos: los grafos sin aristas negativas y los grafos sin ciclos. Como se realizó anteriormente, ya se ha manejado eficientemente grafos con ciclos negativos.

En este problema, se describe cómo resolver el problema de la ruta más corta desde una sola fuente en tiempo lineal en grafos dirigidos acíclicos.

Antes de hacerlo, se necesita realizar una secuencia de actualizaciones (utilizando el algoritmo de Bellman-Ford) que incluya cada ruta más corta como subsecuencia.

En cualquier ruta de un DAG, los vértices aparecen en orden lineal creciente. El DAG busca en profundidad y visita los vértices en orden ordenado, actualizando las aristas fuera de cada uno.

Como dato de entrada, se dispone de un DAG ponderado con pesos de arista enteros de −10^3 a 10^3 y n ≤ 10^5 vértices en el formato de lista de aristas.

El problema requiere entregar un arreglo D[1..n] donde D[i] es la longitud de la ruta más corta desde el vértice 1 hasta el vértice i (D[1]=0). Si i no es alcanzable desde 1, se establece D[i] en x.

- Resolución:

Para la resolución del problema, primeramente, el proceso comienza marcando todos los vértices como no visitados y utilizando la función topologicalSort() para realizar la ordenación topológica. Posteriormente, inicializa una lista de distancias con infinito para cada vértice, excepto para el vértice de origen que se establece en 0. Utiliza una estructura de pila para almacenar los vértices en orden topológico.

Luego, utiliza la pila ordenada topológicamente para actualizar las distancias mínimas desde el vértice de origen hacia los vértices adyacentes, garantizando que la distancia mínima se mantenga. Este proceso continúa hasta que todos los vértices se visiten y se actualicen correctamente.

Finalmente, genera una salida que representa las distancias mínimas desde el vértice de origen hasta cada uno de los vértices del grafo, marcando con "x" aquellos vértices inalcanzables desde el vértice de origen.

El algoritmo que se implemento proporciona una solución eficiente para calcular las distancias mínimas en grafos dirigidos acíclicos, permitiendo el análisis de la conectividad y las rutas más cortas en un grafo con restricciones de dirección y sin ciclos negativos.

https://rosalind.info/problems/sdag/

Bioinformatics Armory

Introduction to the Bioinformatics Armory

- Descripción:

Se plantea la necesidad de conocer programas informáticos que ayuden en el análisis y solución de trabajos en el área de bioinformática ya que así se ahorraría tiempo y esfuerzo en creaciones de programas. Por tanto, es indispensable buscar librerías o programas que resuelvan por nosotros y así nos faciliten el trabajo.

- Resolución:

Este problema se plantea como la necesidad de calcular el recuento de nucleótidos en una secuencia de ADN específica. Es importante destacar que las secuencias de ADN están compuestas por cuatro símbolos: A, C, G y T.

Para resolver este problema, se puede seguir la pista proporcionada en la página de Rosalind utilizando la librería BioPython. Sin embargo, en este caso, optaremos por tratar la secuencia de ADN como una cadena de texto convencional y utilizaremos la función count() para contar las apariciones de cada símbolo en la secuencia. De esta manera, obtendremos el recuento de nucleótidos de forma efectiva.

https://rosalind.info/problems/ini/

GenBank Introduction

- Descripción:

Se anuncia la base de datos de GenBank y su funcionamiento, la cual es fundamental para los biólogos moleculares y otros profesionales que trabajan en genética y biología molecular. Esta base de datos da acceso a mucha información genética, incluyendo secuencias de ADN y proteínas, y datos sobre genes, genomas y otros elementos genéticos.

Los científicos pueden utilizar GenBank para realizar investigaciones, encontrar información sobre genes específicos, comparar secuencias genéticas, estudiar la evolución y realizar una variedad de análisis bioinformáticos.

En el problema se requiere usar la base de datos GenBank, que contiene diversas subdivisiones como Nucleotide, Genome Survey Sequence (GSS) y Expressed Sequence Tags (EST).

Dentro del enunciado se explica cómo realizar búsquedas en la base de datos utilizando consultas de texto general o consultas basadas en nombres de proteínas, genes o símbolos genéticos. También se menciona cómo limitar la búsqueda a ciertos tipos de registros y cómo interpretar los detalles de la búsqueda.

- Resolución:

De los datos de prueba, se obtienen el nombre de un gen y dos fechas en el formato YYYY/M/D, que representan la búsqueda de información dentro de un intervalo específico de tiempo. Con esta información, se requiere obtener el número de entradas o resultados de Nucleotide GenBank para el género especificado que fueron publicados entre las fechas indicadas.

Siguiendo las indicaciones de programación proporcionadas por la página, se utilizó la biblioteca BioPython, la cual incluye un módulo llamado Entrez que permite realizar consultas a la base de datos considerando ciertos parámetros de acuerdo con nuestros criterios.

https://rosalind.info/problems/gbk/

Data Formats

- Descripción:

Se explica que existen varios formatos de presentación de datos utilizados para representar cadenas genéticas. Actualmente, los tres formatos más populares son FASTA, NEXUS y PHYLIP. En este problema, nos familiarizaremos con el formato FASTA.

En el formato FASTA, una cadena se introduce con una línea que comienza con '>', seguida de información etiquetando la cadena. Las líneas siguientes contienen la cadena en sí misma; la siguiente línea que comienza con '>' indica que la cadena actual está completa y comienza la etiqueta de la siguiente cadena en el archivo.

- Resolución:

Para los datos de entrada, se dispone de un conjunto de N IDs de entrada de GenBank, cada uno representando una cadena genética. Se especifica que la cantidad N será menor que 10 IDs en los datos de prueba.

El objetivo del problema es devolver la cadena genética más corta junto con su ID en formato FASTA. Para lograr esto, podemos tratar cada cadena genética en formato FASTA como un objeto utilizando la librería de Python BioPython. Esta librería contiene dos módulos importantes para resolver el problema: Entrez y SeqIO. El primero se encarga de consultar la base de datos de GenBank por medio

de la función efetch() utilizando los IDs proporcionados. El segundo módulo se encarga de organizar las respuestas de la base de datos en una lista.

Una vez obtenidas todas las secuencias genéticas, el objetivo es utilizar la función min() junto con una función lambda para recorrer todas las secuencias y retornar aquella con la menor cantidad de caracteres. De esta manera, obtendremos la cadena genética más corta junto con su respectivo ID en formato FASTA.

https://rosalind.info/problems/frmt/

FASTQ format introduction

- Descripción:

En este ejercicio se detalla cómo se gestiona la calidad de las secuencias genéticas obtenidas mediante secuenciación de alto rendimiento. Estas secuencias se evalúan utilizando la escala de calidad de Phred, donde puntajes más bajos indican menor confiabilidad en la lectura. Estos puntajes se determinan mediante una ecuación específica.

Además, se señala que el formato estándar para almacenar estas secuencias es FASTQ, el cual incluye tanto la secuencia genética como los puntajes de calidad. Cada entrada en un archivo FASTQ se compone de cuatro líneas: la identificación de la secuencia, la secuencia misma, un identificador opcional o comentario, y los puntajes de calidad codificados como símbolos ASCII. Es fundamental que las longitudes de la segunda y cuarta línea coincidan correctamente.

- Resolución:

Se cuenta con un archivo en formato FASTQ que contiene registros de cadenas genéticas. Se solicita realizar una conversión de formato, transformando estos registros al formato FASTA.

Para lograr esta conversión, se puede utilizar la librería BioPython, específicamente el módulo SeqIO, que proporciona la función convert(). Al implementar esta función, es posible transformar el archivo de formato FASTQ a FASTA y viceversa de manera directa y sencilla.

https://rosalind.info/problems/tfsq/

Read Quality Distribution

- Descripción:

Se menciona que características inusuales en los datos pueden indicar problemas en etapas previas, como la preparación de muestras, que deben corregirse para obtener datos válidos.

Se destaca la utilización del puntaje de calidad Phred como una métrica básica en este control. Ya que los archivos FASTQ con estos puntajes pueden ser grandes y complejos.

Dado que una ejecución de secuenciación puede generar miles o millones de lecturas, el enfoque se dirige hacia la evaluación de la calidad del conjunto de datos en su totalidad. Se describe una métrica común en este nivel: la distribución del puntaje de calidad por lectura. Se calcula el promedio de calidad de cada lectura y se analiza la distribución de estos puntajes promedio.

- Resolución:

En los datos de entrada se incluye un número que representa un umbral de calidad, además de una serie de secuencias genéticas en formato FASTQ para múltiples lecturas.

El desafío de este problema consiste en determinar el número de lecturas cuya calidad promedio esté por debajo del umbral proporcionado anteriormente.

Para abordar esta tarea, se empleó el módulo SeqIO de la biblioteca BioPython para manejar el conjunto de secuencias genéticas en formato FASTQ. Se utilizó un bucle for para iterar sobre cada secuencia individualmente, ya que la función parse() devuelve un objeto SeqRecord que contiene los puntajes de calidad de Phred correspondientes a cada base de la secuencia. De esta manera, se accede a los puntajes de calidad y se calcula el promedio. Luego, se verifica si este promedio está por debajo del umbral, sumándole una unidad a un contador establecido previamente que representa el número de lecturas que cumplen con la condición.

https://rosalind.info/problems/phre/

Protein Translation

- Descripción:

Se describe el proceso de traducción de una secuencia de nucleótidos en una región codificante del genoma en una cadena de aminoácidos. Se explica que, para lograr esta traducción, se utilizan los códigos genéticos que convierten los tripletes de nucleótidos en aminoácidos. Se mencionan los codones de inicio (AUG) y de parada (UAA, UAG, UGA), así como la variación del código genético que algunos

organismos y organelos utilizan. También se destaca la importancia de verificar la fuente de los datos genómicos antes de realizar la traducción.

- Resolución:

El problema consiste en determinar el índice de la variante del código genético que se utilizó para traducir una cadena de ADN en una cadena de proteínas. En los datos de entrada se proporciona una cadena de ADN y su traducción correspondiente en una cadena de proteínas.

El objetivo es encontrar el índice de la tabla de codones genéticos que se utilizó para realizar esta traducción. Es importante destacar que, si existen múltiples soluciones, cualquiera de ellas se considerará válida para la solución del problema.

Para resolver este problema, se emplearon dos módulos de BioPython. El primero es CodonTable, que mediante su atributo “ambiguous_generic_by_id" muestra todos los índices de la tabla de codones. Esto permite iterar sobre cada uno de ellos utilizando un bucle for y validar cuál de todas las tablas puede corresponder con la traducción.

El segundo módulo es la función translate(), la cual recibe como parámetros la cadena de ADN y el índice de la tabla que se está iterando en ese momento. Esta función proporciona la traducción de la proteína, que se compara con la cadena de proteína de los datos de entrada. Si coinciden, se afirma que ese índice de la tabla de codones es válido para la traducción. Este proceso se repite para todas las tablas de codones disponibles, y los resultados se guardan en una lista para luego devolver cualquiera de las soluciones obtenidas.

https://rosalind.info/problems/ptra/

Read Filtration by Quality

- Descripción:

El problema aborda la importancia de filtrar las lecturas de baja calidad en una secuenciación, lo cual puede tener múltiples beneficios, como mejorar los resultados y reducir el tiempo y la memoria necesarios para el análisis o ensamblaje.

- Resolución:

Se proporciona un umbral de calidad, un porcentaje de bases y un conjunto de entradas FASTQ.

En el problema, se debe determinar el número de lecturas en las entradas FASTQ filtradas utilizando alguna herramienta de filtrado de calidad FASTQ del kit de

herramientas FASTX. Sin embargo, se tomó otro enfoque, reutilizando lo aprendido anteriormente en el problema de la Read Quality Distribution.

Sabiendo que por cada secuencia genética dada por los objetos SeqRecord de la función parse, se pueden obtener una lista de los puntajes de calidad de Phred correspondientes a cada base de la secuencia. Lo primero que se debe hacer es filtrar por aquellos puntajes que sean iguales o superiores al umbral de calidad dado, así se creará otra lista con los puntajes aceptados necesarios.

Seguidamente, se debe calcular y verificar si el porcentaje equivalente de la nueva lista de los puntajes filtrados es mayor o igual al porcentaje de bases establecido en los datos de entrada, todo eso efectuando una simple regla de tres con ayuda de la función len() para obtener la cantidad de elementos de una lista.

Con anterioridad, se realizó la inicialización de una variable llamada contador, además de realizarse el parseo de las secuencias genéticas, para que, dentro de un bucle for, se realice todo lo mencionado anteriormente.

Finalmente, al cumplirse el condicional del porcentaje obtenido de la lista filtrada de puntajes, se sumará una unidad al contador. El bucle se repetirá hasta iterar sobre la última secuencia genética y se devolverá el valor acumulado por el contador como resultado del problema.

https://rosalind.info/problems/filt/

Complementing a Strand of DNA

- Descripción:

Se describe la estructura del ADN, destacando que está formado por dos hebras que corren en direcciones opuestas. Cada base de una hebra se une con una base complementaria en la otra hebra: adenina siempre con timina, y citosina siempre con guanina. Estas dos hebras están torcidas juntas formando una estructura en forma de doble hélice. Como el ADN es de doble hebra, durante el análisis de secuencias se debe examinar tanto la cadena de ADN dada como su complemento inverso.

- Resolución:

El problema planteado consiste en encontrar cuántas de las cadenas de ADN dadas coinciden con sus complementos inversos. Para ello, se obtiene una colección de secuencias de ADN de cantidad no mayor a 10 secuencias por colección.

Para resolver el problema, se crea una variable que lleve la cuenta de cuántas cadenas de ADN cumplen con la condición dada en el ejercicio. Luego, se parsea

el archivo en formato FASTA para obtener un conjunto de objetos SeqRecords para el manejo de funciones de la librería BioPython.

Obteniendo el conjunto de objetos, solo se debe utilizar un bucle for para iterar sobre cada secuencia de ADN y, usando la función reverse_complement(), obtener su complemento inverso y así mismo, utilizar una condición para validar la igualdad.

Al validarse la igualdad de una secuencia de ADN con su complemento inverso, el contador suma una unidad y, finalizando el bucle, se devuelve el valor del contador como resultado.

https://rosalind.info/problems/rvco/

Base Quality Distribution

- Descripción:

Este problema presenta las métricas de calidad para el control de calidad (QC) de datos de secuenciación. Una métrica simple es la calidad de llamada de base, que indica la calidad promedio de cada base en una secuencia.

Se muestra un ejemplo de un conjunto de datos de buena calidad y otro de mala calidad, donde se observa una distribución más amplia de la calidad en el segundo caso. Se menciona que la calidad de las bases puede variar según su posición en la lectura.

- Resolución:

El problema planteado consiste en contar el número de posiciones en las que la calidad promedio de la base cae por debajo de un umbral dado en un archivo FASTQ. Por tanto, dentro de los datos de entrada se encuentra el valor del umbral de calidad seguido de los registros de secuencias genéticas en formato FASTQ.

Primeramente, se optó por utilizar una matriz NxM como estructura de datos principal para guardar la información necesaria, dado que de las N secuencias genéticas del FASTQ, existirán por cada una de ellas, un listado de puntajes de calidad de cantidad M. Estos puntajes serán obtenidos de la misma manera que se hizo en el problema "Read Quality Distribution".

Se implementó la estructura de la matriz ya que, para obtener la calidad de base media de cada posición, se creó una nueva lista que contiene los resultados de las sumatorias de los valores de M_i posición de todos los N.

Luego de eso, por cada resultado dentro de la nueva lista, se verificará por medio de una proporcionalidad con el número de secuencias, si el resultado cae por debajo del umbral de calidad establecido. Si el resultado es menor que el umbral de calidad,

se suma una unidad en un contador, hasta terminar todas las sumatorias de las posiciones de las secuencias genéticas dadas, finalizando con el resultado del número de posiciones que caen en el contador.

https://rosalind.info/problems/bphr/

Finding Genes with ORFs

- Descripción:

Se describe la clasificación de marcos de lectura abiertos (ORF por sus siglas en inglés) en secuencias de ADN. Un ORF es un marco de lectura que comienza con un codón de inicio (ATG) y termina con un codón de parada (TAA, TAG o TGA), sin tener otros codones de parada entre ellos.

Se menciona que hay seis marcos de lectura para cualquier hebra de ADN, tres derivados de la traducción de la hebra misma y tres de cambios a la hebra complementaria. Por tanto, identificar genes mediante la búsqueda de ORFs es una simplificación, ya que en organismos más complejos como eucariotas pueden requerir la identificación de promotores e intrones. Sin embargo, en procariotas y virus, cuyos genomas son menos complicados, usar ORFs para identificar genes es una aproximación útil.

- Resolución:

El problema consiste en hallar la cadena de proteínas más extensa que pueda ser traducida a partir de un marco de lectura abierta (ORF, por sus siglas en inglés) en una secuencia de ADN dada, asumiendo el código genético estándar para la traducción de ARN a proteínas. Para ello, como dato de entrada se dispone de una secuencia de ADN que consta de 1000 pares de bases.

Mediante el uso de Biopython, es posible encontrar ORFs empleando los métodos translate() y reverse_complement() del módulo Bio.Seq, utilizando los objetos Seq. No obstante, se presenta un inconveniente al traducir la secuencia de ADN, ya que, en las tablas de codones, existen una o varias traducciones para un mismo trío de aminoácidos de la secuencia.

Para abordar la resolución del problema, se propone la creación de una función denominada finding_genes_with_orfs() que, en primer lugar, convierte la cadena de ADN en un objeto Seq(). Luego, obtiene la secuencia inversa complementaria. Después, busca ORFs en ambas secuencias (original y complementaria) comenzando desde las tres posiciones de inicio posibles (0, 1, 2). Para cada inicio potencial, traduce la secuencia a una cadena de aminoácidos utilizando translate(), especificando un símbolo de parada como "*"; posteriormente, divide esta cadena en subcadenas en cada "*".

Una vez finalizada la traducción, la función filtra estas subcadenas para encontrar aquellas que contienen el codón de inicio "M" (Metionina) y guarda la subcadena más larga encontrada. Al concluir, devuelve la cadena de aminoácidos más extensa identificada, correspondiente al ORF más largo en la secuencia de ADN.

https://rosalind.info/problems/orfr/

Base Filtration by Quality

- Descripción:

En el problema se habla sobre cómo manejar letras de mala calidad en las lecturas de secuenciación de ADN. En lugar de filtrar las lecturas completas (como se hace en "Read Filtration by Quality"), se sugiere recortar las bases de baja calidad de ambos extremos de las lecturas. Esto ayuda a conservar una gran cantidad de información.

Se plantea que las bases de mala calidad pueden recortarse fácilmente utilizando ciertos umbrales (definidos por un gráfico de calidad similar a lo que se hizo en "Base Quality Distribution").

Tambien se mencionan algunas herramientas para realizar este recorte, como FASTQ Quality Trimmer en Galaxy y Trimmomatic. Se especifica que, para un recorte simple de ambos extremos, se deben establecer parámetros como el tamaño de la ventana y los valores de calidad límite.

- Resolución:

El problema implica tomar un valor de corte de calidad y la manipulación de un archivo FASTQ, con el propósito de producir un archivo FASTQ modificado que recorte desde ambos extremos, eliminando las bases iniciales y finales con una calidad inferior al valor especificado.

Para abordar este problema, en primer lugar, se emplea la librería Biopython para leer el archivo FASTQ. A través de la función parse(), se obtiene una colección de secuencias representadas como objetos SeqRecord. Posteriormente, se itera sobre cada una de estas secuencias mediante un bucle for, extrayendo los puntajes de calidad mediante el atributo letter_annotations["phred_quality"].

Utilizando dos bucles while, se determinan los índices que señalan el inicio y el final de cada secuencia, refiriéndose así a sus extremos. Se establece una comparación en cada extremo para verificar si el puntaje de calidad es superior al umbral especificado. Si se encuentra un puntaje menor al umbral, se ajustan los límites del extremo correspondiente, lo que conduce al recorte progresivo de la secuencia.

Una vez recortadas las bases de baja calidad, las secuencias resultantes se agregan al objeto OUTPUT, que almacena todas las secuencias modificadas en formato FASTQ.

https://rosalind.info/problems/bfil/

Bioinformatics Stronghold

Counting DNA Nucleotides

- Descripción:

Se habla sobre la célula como la unidad básica de la vida y el núcleo como una parte importante de las células eucariotas, que está lleno de una sustancia llamada cromatina, que se condensa en cromosomas durante la mitosis. Se menciona que uno de los componentes de la cromatina son los ácidos nucleicos, que son polímeros formados por unidades llamadas nucleótidos. Cada nucleótido está compuesto por un azúcar, un fosfato y una base nitrogenada.

El ADN es un tipo de ácido nucleico que consiste en cadenas de nucleótidos con cuatro bases: adenina (A), citosina (C), guanina (G) y timina (T). Se destaca que el ADN se encuentra en todos los organismos vivos y se reserva el término "genoma" para referirse al total del ADN en los cromosomas de un organismo.

- Resolución:

El problema consiste en contar el número de veces que aparecen los símbolos 'A', 'C', 'G' y 'T' en una cadena de ADN, proporcionada con una longitud menor a 1000 caracteres. Para abordar este problema en Python, podemos aprovechar la función nativa count(), la cual, como su nombre en inglés indica, cuenta el número de apariciones de un tipo de dato dentro de una secuencia, ya sea una cadena, números, listas, tuplas, etc.

La estrategia para resolver este problema implica utilizar un bucle for para iterar sobre las cuatro bases mencionadas anteriormente e implementar la función count() para obtener el número de apariciones de cada base en la cadena de ADN. Es importante mencionar que, en los problemas de Python Village, se destacó que las cadenas comparten funcionalidades o comportamientos con las listas, ya que se considera que una cadena es una lista ordenada de caracteres.

Finalmente, almacenamos los resultados en una variable previamente definida. Al concluir el bucle for, devolvemos el resultado obtenido de la cuenta de las cuatro bases en la cadena de ADN.

https://rosalind.info/problems/dna/

Transcribing DNA into RNA

- Descripción:

Se menciona la existencia de un segundo ácido nucleico llamado ácido ribonucleico o ARN, que es diferente del ADN en varios aspectos, incluyendo la presencia de una base llamada uracilo en lugar de timina y la presencia de un azúcar diferente

llamado ribosa. Se describe que el ARN se encuentra en todas las células de los organismos vivos, al igual que el ADN. Además, se explica que el ARN se forma a partir de una molécula de ADN durante un proceso llamado transcripción, donde se usa una cadena de ADN como plantilla para construir una cadena de ARN, reemplazando la timina con uracilo.

- Resolución:

El problema planteado consiste en obtener la cadena de ARN transcrita a partir de una cadena de ADN dada, sustituyendo todas las ocurrencias de 'T' por 'U'. En Python, se dispone de la función replace(), la cual reemplaza un elemento específico en un texto. Como su nombre en inglés indica, esta función sustituye el elemento mencionado por otro especificado en la función.

Por lo tanto, la solución es directa: implementamos la función replace() indicando los valores de "T" y "U", y así obtenemos la cadena de ARN ya transcrita.

https://rosalind.info/problems/rna/

Complementing a Strand of DNA

- Descripción:

Se explica sobre las estructuras secundaria y terciaria del ADN. Se menciona que la estructura primaria del ADN se refiere a la disposición de las bases nitrogenadas a lo largo de la cadena de nucleótidos. Sin embargo, para entender completamente el ADN, es crucial comprender su forma tridimensional.

Se hace referencia al trabajo de James Watson y Francis Crick, quienes propusieron la estructura del ADN en 1953. Según su modelo, el ADN está formado por dos hebras que corren en direcciones opuestas y se unen mediante pares de bases complementarias: adenina con timina, y citosina con guanina. Estas interacciones entre bases forman la estructura secundaria del ADN, mientras que la estructura tridimensional se refiere a la forma de doble hélice.

- Resolución:

El problema planteado consiste en encontrar el complemento reverso de una cadena de ADN, lo cual implica revertir la cadena original y luego tomar el complemento de cada base. Por ejemplo, el complemento reverso de "GTCA" sería "TGAC". Se recibe como dato de entrada una cadena de ADN con una longitud de hasta 1000 pares de bases.

Para abordar esta situación, se ha desarrollado una función llamada reverse_complement(), que acepta una cadena de ADN como entrada y devuelve su complemento inverso.

El proceso de la función comienza inicializando una cadena vacía. Luego, recorre la cadena de ADN de derecha a izquierda, identificando el complemento de cada base de ADN utilizando un diccionario de mapeo predefinido. Este diccionario asocia cada base de ADN con su complemento correspondiente: A se complementa con T, C con G, G con C y T con A. Una vez encontrado el complemento de cada base, se agrega a la cadena vacía. Finalmente, la función retorna la cadena resultante, que representa el complemento inverso de la cadena de ADN original.

https://rosalind.info/problems/revc/

Rabbits and Recurrence Relations

- Descripción:

El problema se basa en un ejercicio matemático propuesto por Leonardo de Pisa, también conocido como Fibonacci, que involucra el crecimiento de una población de conejos. En el ejercicio, se establecen ciertas suposiciones simplificadas sobre la reproducción de los conejos, como que comienzan con una pareja de conejos recién nacidos, que los conejos alcanzan la edad reproductiva después de un mes, y que cada pareja de conejos en edad reproductiva se reproduce cada mes, produciendo una nueva pareja de conejos. El objetivo es calcular cuántas parejas de conejos habrá después de un año.

Se menciona que, aunque las suposiciones de inmortalidad de los conejos pueden parecer exageradas, en un entorno sin depredadores reales, como cuando los conejos europeos fueron introducidos en Australia en el siglo XIX, estas suposiciones no eran tan poco realistas. La introducción de conejos en Australia tuvo graves repercusiones ambientales, incluida la degradación del ecosistema y la erosión de grandes áreas de terreno.

- Resolución:

El problema consiste en calcular el número total de parejas de conejos que habrá después de n meses, dado que, en cada generación, cada pareja de conejos reproductores produce una camada de k parejas de conejos. Donde n es un entero positivo menor o igual a 40, y k es un entero positivo menor o igual a 5.

Para resolución del problema, se desarrolló una función que calcula el número de conejos en una población después de un cierto número de meses, utilizando un enfoque de programación dinámica.

Si el número de meses es 0, devuelve 0, indicando que no hay conejos en ese momento. Si es 1, devuelve 1, indicando que hay un conejo en ese momento.

Para meses mayores a 1, la función inicializa una lista llamada generations que almacenará el número de conejos en cada generación. Luego, calcula el número de

conejos en cada generación utilizando un bucle for, donde para cada mes i desde 2 hasta months, se calcula sumando el número de conejos en la generación anterior con el número de conejos que se reproducen en esa generación.

Finalmente, la función devuelve el número de conejos en la generación months, permitiendo así calcular el crecimiento de la población de conejos de manera eficiente y evitando recalcular los mismos valores varias veces mediante el uso de la programación dinámica.

https://rosalind.info/problems/fib/

Computing GC Content

- Descripción:

El método rápido utilizado por los primeros programas informáticos para determinar el idioma de un texto dado era analizar la frecuencia con la que aparecía cada letra en el texto. Esta estrategia se usaba porque cada idioma exhibió sus frecuencias de letras, y si el texto considerado es lo suficientemente largo, el software reconocerá correctamente el idioma con rapidez y una tasa de error muy baja.

Una aplicación biológica similar surge al encontrar una molécula de ADN de una especie desconocida. Debido a las relaciones de apareamiento de bases de las dos cadenas de ADN, la citosina y la guanina siempre aparecerán en cantidades iguales en una molécula de ADN de doble cadena. Por lo tanto, para analizar las frecuencias de símbolos de ADN para compararlas con una base de datos, calculamos el contenido de GC de la molécula, es decir, el porcentaje de sus bases que son citosina o guanina.

En la práctica, el contenido de GC de la mayoría de los genomas eucariotas ronda el 50%. Como los genomas son tan largos, podemos distinguir especies según discrepancias muy pequeñas en el contenido de GC; además, la mayoría de los procariontes tienen un contenido de GC significativamente mayor que el 50%, por lo que el contenido de GC se puede utilizar para diferenciar rápidamente muchos procariontes y eucariontes usando muestras de ADN relativamente pequeñas.

- Resolución:

Se plantea calcular el contenido de GC de una cadena de ADN, dado el porcentaje de símbolos en la cadena que son 'C' o 'G'. Además, se menciona el formato FASTA utilizado para etiquetar las cadenas de ADN en una base de datos. Para ello, se tienen un conjunto de datos de entrada conformados por al menos 10 cadenas de ADN en formato FASTA y, se desea obtener el ID de la cadena con el contenido de GC más alto, seguido por el contenido de GC de esa cadena.

Primeramente, mediante el empleo de la biblioteca SeqIO del paquete Biopython, el código realiza la lectura de una secuencia de ADN contenida en un archivo en formato FASTA.

Luego, se lleva a cabo el cálculo del contenido de GC para cada secuencia de ADN presente en el archivo FASTA. Este proceso implica la iteración a través de cada secuencia de ADN en el archivo, contando el número de ocurrencias de las bases "G" y "C" y calculando su proporción respecto al total de bases.

Utilizando una variable temporal, podemos determinar de manera eficiente el contenido de GC máximo en las secuencias de ADN. Esta variable se inicializa con un valor predeterminado y se actualiza cada vez que se encuentra un contenido de GC mayor mientras se recorren las secuencias del archivo FASTA.

De esta manera, al concluir el análisis de todas las secuencias, la variable contendrá el ID de la secuencia con el contenido de GC más alto.

https://rosalind.info/problems/gc/

Counting Point Mutations

- Descripción:

Las mutaciones son errores que pueden ocurrir durante la creación o copia de ácidos nucleicos, especialmente el ADN, y pueden tener efectos significativos en las células debido a la importancia de estos ácidos en las funciones celulares. Aunque se consideran errores, algunas mutaciones pueden ser beneficiosas y contribuir a la evolución de las especies durante generaciones. Una mutación puntual, que implica el reemplazo de una base por otra en un solo nucleótido, es el tipo más común de mutación de ácido nucleico.

La comparación de cadenas de ADN homólogas de diferentes organismos permite estimar el número mínimo de mutaciones puntuales que podrían haber ocurrido a lo largo de su evolución. Este análisis es fundamental para comprender las relaciones evolutivas entre especies y minimizar el número de mutaciones para explicar las historias evolutivas de manera simple, siguiendo el principio de parsimonia biológica.

- Resolución:

El problema consiste en calcular la distancia de Hamming entre dos cadenas de ADN, s y t, que tienen la misma longitud y no exceden los 1000 pares de bases. La distancia de Hamming se define como el número de posiciones en las que los símbolos correspondientes de las cadenas s y t son diferentes. En otras palabras, se busca determinar cuántos nucleótidos difieren entre las dos cadenas de ADN en las mismas posiciones.

Para la resolución, se creó una función llamada counting_mutations() que calcula la distancia de Hamming entre dos cadenas de ADN.

Primero, la función verifica si las dos cadenas de ADN tienen la misma longitud. Si no lo hacen, devuelve un error. Esto garantiza que las cadenas de ADN proporcionadas para su comparación tengan la misma longitud.

Luego, la función utiliza la función zip() para iterar simultáneamente sobre los caracteres de las dos cadenas de ADN. En cada iteración, compara los caracteres correspondientes de las dos cadenas usando una condicional de comparación. Si

Si los caracteres son diferentes, la condicional lo evaluará verdadero, lo que contribuirá con un 1 a la suma total. Si son iguales, la condicional evaluará como falso, lo que no contribuirá con nada a la suma total.

Finalmente, la función devuelve la suma total, que representa la distancia de Hamming entre las dos cadenas de ADN.

https://rosalind.info/problems/hamm/

Mendel's First Law

- Descripción:

Se introduce los conceptos básicos de la herencia mendeliana y la probabilidad en genética. En cuanto a la herencia, se menciona la ley de segregación de Mendel, que explica cómo los organismos heredan pares de alelos de sus padres, y la diferencia entre alelos dominantes y recesivos. Se describe el uso de cuadros de Punnett para predecir los resultados de la herencia.

En relación con la probabilidad, se explica cómo se modelan fenómenos aleatorios utilizando variables aleatorias y se introducen los conceptos de eventos y diagramas de árbol de probabilidad.

- Resolución:

El problema planteado requiere calcular la probabilidad de que dos organismos seleccionados al azar produzcan un descendiente con un alelo dominante. Esta probabilidad se fundamenta en el número de organismos homocigotos dominantes, heterocigotos y homocigotos recesivos presentes en la población, representados por tres enteros positivos: K, M y N, respectivamente. Es crucial destacar que la suma de estos tres grupos de organismos constituye la población total. Además, se parte del supuesto de que cualquier par de organismos puede aparearse para la reproducción.

Para abordar esta problemática, se ha implementado una función denominada firts_law_medel_2() que calcula la probabilidad de que un organismo seleccionado al azar de una población siga un patrón de herencia mendeliana, conforme a la Primera Ley de Mendel.

Para llevar a cabo este cálculo, la función emplea la fórmula del modelo mendeliano para determinar la probabilidad de obtener un fenotipo dominante. En primer lugar, se define una función interna denominada take_two(n) que calcula el número de pares posibles que se pueden formar a partir de n organismos utilizando la fórmula combinatoria.

Posteriormente, se calcula el total de pares posibles entre todos los organismos al utilizar la función take_two() y sumar la cantidad de cada tipo de organismo.

Seguidamente, se determina el número de pares posibles que resultan en el fenotipo dominante conforme al modelo mendeliano. Esto implica considerar diversas combinaciones de homocigotos dominantes, heterocigotos y homocigotos recesivos.

Por último, se estima la probabilidad al dividir el número de pares dominantes entre el total de pares posibles y se retorna este valor como resultado.

https://rosalind.info/problems/iprb/

Translating RNA into Protein

- Descripción:

En el problema, se explica que las proteínas, compuestas por aminoácidos, son esenciales para las funciones celulares y son clave para entender la vida. Se describe cómo el código genético traduce moléculas de ARN, específicamente el ARN mensajero (ARNm), en aminoácidos para la creación de proteínas. Se mencionan los codones de ARN que especifican la traducción de secuencias de nucleótidos en aminoácidos y se destaca la importancia de los codones de inicio y parada en la traducción. Además, se presenta el dogma central de la biología molecular, que establece que las proteínas siempre se crean a partir de ARN, que a su vez siempre se crea a partir de ADN, como una aproximación fundamental a la verdad en biología molecular.

Igualmente, se describe cómo los ribosomas crean péptidos utilizando ARN de transferencia (ARNt), y cómo estos péptidos forman cadenas de aminoácidos para construir proteínas. Se señala que los genes, que son intervalos de ADN o ARN que codifican proteínas, son fundamentales para la diferenciación de los organismos y la herencia de los rasgos.

- Resolución:

El problema se centra en recibir como dato de entrada una secuencia de ARN correspondiente a una molécula de ARNm y posteriormente devolver la cadena de proteína codificada por dicha secuencia de ARN.

Para resolver este problema, se consideraron dos enfoques. El primero consiste en utilizar la biblioteca Biopython para convertir la secuencia de ARN en un objeto SeqRecord y emplear su función translate(to_stop=True). De esta manera, se guarda el resultado en una nueva variable y luego se devuelve como solución al problema.

Por otro lado, la segunda forma de abordar el problema fue desarrollando una función llamada translating_protein(). Esta función traduce una cadena de ARN en una secuencia de proteínas utilizando un diccionario de codones de ARN y la cadena de ARN proporcionada como entrada. Posteriormente, itera sobre la cadena de ARN en segmentos de tres nucleótidos (codones), buscando en el diccionario el correspondiente aminoácido para cada codón. Si encuentra un aminoácido válido (diferente a "Stop"), lo agrega a la cadena de proteínas que se está construyendo. Finalmente, devuelve la cadena de proteínas resultante.

https://rosalind.info/problems/prot/

Finding a Motif in DNA

- Descripción:

Se menciona la importancia de encontrar intervalos idénticos de ADN en los genomas de dos organismos diferentes, lo que sugiere que estos intervalos tienen la misma función en ambos organismos. Estos intervalos compartidos se denominan "motivos" y son objeto de búsqueda común en la biología molecular.

Sin embargo, la tarea se complica debido a que los genomas están llenos de intervalos de ADN que se repiten múltiples veces, conocidos como repeticiones. Estas repeticiones ocurren con mucha más frecuencia de lo que se esperaría por azar, lo que indica que los genomas no son aleatorios y que el lenguaje del ADN debe ser muy poderoso. También, se menciona la repetición Alu, la más común en humanos, que se repite aproximadamente un millón de veces en cada genoma humano.

- Resolución:

El problema requiere encontrar todas las ubicaciones de una subcadena de ADN (t) dentro de otra cadena de ADN (s). Esto se realiza identificando todas las posiciones donde la subcadena t aparece como una colección contigua de símbolos dentro de s.

Para la solucion del problema, se creó desarrollo una función llamada finding_motif_in_dna() que busca ocurrencias de un motivo específico dentro de una secuencia de ADN proporcionada.

Esta función acepta dos parámetros de tipo cadena: una cadena S, que representa la secuencia de ADN, y una cadena T, que es el motivo que se busca dentro de esa secuencia. La función devuelve una cadena que contiene las posiciones donde se encuentra el motivo en la secuencia de ADN.

Para realizar la búsqueda, la función calcula las longitudes de las cadenas S y T. Luego, itera a través de la secuencia de ADN utilizando un bucle for que abarca todos los posibles puntos de inicio para el motivo. En cada iteración, examina una subcadena de la secuencia de ADN que comienza en la posición actual del bucle y tiene la misma longitud que el motivo buscado.

Si esta subcadena coincide exactamente con el motivo, la función registra la posición donde se encontró el motivo en una cadena llamada locations. Finalmente, devuelve esta cadena locations que contiene las posiciones donde se encontraron ocurrencias del motivo en la secuencia de ADN.

https://rosalind.info/problems/subs/

Consensus and Profile

- Descripción:

Se describe el problema de encontrar un ancestro común más probable al analizar varias cadenas de ADN homólogas simultáneamente. Para ello, se utiliza una matriz de perfil que representa la frecuencia de cada base en cada posición de las cadenas de ADN. A partir de esta matriz, se obtiene una cadena de consenso, que es una cadena formada por los símbolos más comunes en cada posición de las cadenas originales. Este proceso implica encontrar el símbolo más común en cada columna de la matriz de perfil.

- Resolución:

El problema consiste en recibir una colección de cadenas de ADN de igual longitud y devolver una cadena de consenso junto con la matriz de perfil correspondiente. Si existen varias cadenas de consenso posibles, se puede devolver cualquiera de ellas. La colección de cadenas de ADN vendrá como datos de entrada en formato FASTA.

Para la solución, se utilizó la librería de Biopython para trabajar con las secuencias de ADN y utilizar varias herramientas que optimizan el trabajo.

Teniendo la colección de secuencias de ADN en un listado con ayuda de la función SeqIO.parse(), se utilizó la función motifs.create() para crear un objeto de motivos a partir de las secuencias de ADN. Este objeto contiene información sobre la frecuencia de cada base en cada posición de las secuencias.

Luego, se genera un resumen de los datos obtenidos, incluyendo la secuencia de consenso (la secuencia más común encontrada en las posiciones alineadas de las secuencias de ADN) y una tabla de recuento de las bases observadas en cada posición. Este resumen se almacena en una variable llamada output.

El resumen se genera mediante la manipulación de los datos obtenidos del objeto de motivos. Se extraen los recuentos de bases y se formatean para que coincidan con el formato deseado, eliminando los decimales y los espacios en blanco innecesarios. Estos datos formateados se agregan al output y se devuelven como solución al problema.

https://rosalind.info/problems/cons/

Mortal Fibonacci Rabbits

- Descripción:

Se presenta el problema de la proliferación de conejos introducidos en Australia y la posterior construcción de una cerca para contenerlos. La situación llegó a ser tan grave que, incluso después de la construcción de la cerca, se tuvo que recurrir a la liberación de un virus para controlar la población de conejos. Sin embargo, este virus también encontró resistencia en los conejos.

El problema presentado implica expandir el modelo de población de conejos de Fibonacci para tener en cuenta la mortalidad de los conejos.

- Resolución:

Se propone modificar la relación de recurrencia original para considerar que los conejos fallecen después de un número fijo de meses.

Con este fin, se ha implementado la función mortal_fibonacci_rabbits_3(), la cual calcula la cantidad de conejos que estarán presentes después de n generaciones en una población de conejos siguiendo un modelo de Fibonacci mortal.

En primer lugar, se establecen las dos primeras generaciones de conejos en la lista generations. Luego, mediante un bucle while, se calcula cada generación subsiguiente basada en la regla de recurrencia de la secuencia de Fibonacci hasta alcanzar la generación deseada n.

Es importante señalar que la regla de recurrencia se modifica al llegar a la generación m. Antes de este punto, el cálculo es sencillo y se basa en la secuencia de Fibonacci clásica. Sin embargo, a partir de la generación m, los conejos empiezan a fallecer. Por lo tanto, la regla de recurrencia se ajusta para tener en cuenta estas defunciones, restando el número de conejos que perecieron en la generación (m+1) anterior.

El resultado final es la cantidad de conejos presentes en la generación n, la cual se devuelve como el último elemento de la lista generations. Este valor representa el tamaño de la población de conejos después de n generaciones en el modelo de Fibonacci mortal.

https://rosalind.info/problems/fibd/

Overlap Graphs

- Descripción:

Se hace una introducción sobre los conceptos básicos de teoría de grafos, que es el estudio matemático abstracto de redes compuestas por nodos y bordes. Se explica la terminología utilizada en la teoría de grafos, como nodos, bordes, grados, caminos, ciclos y distancias entre nodos.

El problema implica construir un grafo de superposición a partir de una colección de secuencias de ADN en formato FASTA.

En este grafo, cada secuencia se representa como un nodo, y dos secuencias están conectadas por un borde dirigido si hay una superposición de sufijo de longitud k de una secuencia que coincide con un prefijo de longitud k de otra secuencia, siempre que las secuencias sean diferentes entre sí.

- Resolución:

El objetivo es devolver la lista de adyacencia correspondiente al grafo de superposición con k igual a 3.

Para ello, se desarrolló la función Overlap_Graphs() genera un grafo de superposición entre secuencias de ADN en un conjunto dado. Acepta una lista de objetos de secuencia de ADN (llamada dna_collections) y un parámetro k que determina la longitud de la superposición deseada entre las secuencias.

Para cada par de secuencias en la lista dna_collections, la función compara los últimos k caracteres de la primera secuencia con los primeros k caracteres de la segunda secuencia. Si hay una coincidencia, lo que indica una superposición, se registra la conexión en un formato específico. Este formato consiste en una línea

que enumera los identificadores de las dos secuencias que se superponen, separados por un espacio en blanco.

La función acumula todas estas conexiones en una cadena de texto output, que luego devuelve como resultado. Esta cadena contiene todas las conexiones de superposición entre las secuencias de ADN en el conjunto proporcionado.

https://rosalind.info/problems/grph/

Calculating Expected Offspring

- Descripción:

Se destaca la importancia de los promedios en diversas áreas, desde deportes hasta biología molecular. En este problema, se aborda la necesidad de calcular el valor esperado de un proceso aleatorio, que representa el promedio a largo plazo de una variable aleatoria a lo largo de un gran número de ensayos.

Se explica que el valor esperado de una variable aleatoria se calcula como la suma de cada valor posible multiplicado por su probabilidad correspondiente.

- Resolución:

El problema requiere calcular el número esperado de descendientes que exhiben el fenotipo dominante en la próxima generación, dados los números de parejas que tienen cada combinación genotípica. Se asume que cada pareja tiene exactamente dos descendientes.

Para ello, se implementó una función llamada calculating_expected_offspring() que estima la cantidad esperada de descendientes con un fenotipo dominante en la próxima generación, dadas las probabilidades de herencia para diferentes genotipos de parejas en una población.

Los datos de entrada son seis enteros no negativos, cada uno no supera los 20.000. Estos enteros representan el número de parejas en la población que poseen cada combinación genotípica para un factor dado. Específicamente, los seis enteros representan el número de parejas con los siguientes genotipos: AA-AA, AA-Aa, AA-aa, Aa-Aa, Aa-aa y aa-aa, respectivamente.

La función devuelve la cantidad esperada de descendientes con el fenotipo dominante en la próxima generación, bajo la suposición de que cada pareja tiene exactamente dos descendientes. Este cálculo se realiza multiplicando el número de parejas por la probabilidad de que su descendencia tenga el fenotipo dominante, sumando estos productos para todas las combinaciones de parejas y probabilidades.

https://rosalind.info/problems/iev/

Finding a Share Motif

- Descripción:

Se presenta el problema de buscar a través de varias secuencias genéticas en busca de una subcadena común más larga. Mientras que en problemas anteriores se conocía de antemano la subcadena a buscar, en este caso los biólogos deben buscar regiones de similitud entre varias secuencias genéticas para identificar posibles genes compartidos por diferentes organismos o especies.

El problema consiste en encontrar la subcadena común más larga entre una colección de secuencias genéticas. Se define una subcadena común como aquella que aparece en todas las secuencias de la colección, y una subcadena común más larga como aquella que no puede ser extendida sin dejar de ser común.

- Resolución:

Se busca encontrar una única subcadena común más larga entre las secuencias proporcionadas, aunque puede que no sea única y existan múltiples soluciones posibles.

Se desarrollo una función llamada finding_shared_motif1() que encuentra y devuelve el motivo compartido más largo entre una colección de secuencias de ADN. Acepta una lista de secuencias de ADN (llamada dna_collection) como entrada.

Internamente, la función define dos funciones auxiliares:

1. search_substrings(): Esta función encuentra todas las subcadenas posibles dentro de una cadena dada. Itera sobre la cadena de entrada y, para cada longitud posible de subcadena, crea un conjunto de todas las subcadenas de esa longitud. Luego, agrega estos conjuntos a una lista de salida y reduce la longitud de las subcadenas en cada iteración hasta alcanzar subcadenas de longitud 1.

2. if_substr_in_all(): Esta función verifica si una subcadena dada está presente en todas las secuencias de ADN de la lista dada. Itera sobre la lista de secuencias y devuelve falso si encuentra alguna secuencia que no contenga la subcadena dada. Si todas las secuencias contienen la subcadena, devuelve verdadero.

La función principal ordena las secuencias de ADN de menor a mayor longitud, luego selecciona la secuencia más corta como punto de referencia. Utiliza la función

search_substrings() para encontrar todas las subcadenas posibles en esta secuencia corta.

Luego, para cada subcadena encontrada, utiliza la función if_substr_in_all() para verificar si está presente en todas las secuencias de ADN, excepto la primera (ya que la primera es la secuencia de referencia). Tan pronto como encuentra una subcadena que está presente en todas las secuencias, la devuelve como el motivo compartido más largo.

Si ninguna subcadena es compartida por todas las secuencias de ADN, la función devuelve una cadena vacía.

https://rosalind.info/problems/lcsm/

Independent Alleles

- Descripción:

En este problema se aborda la segunda ley de Mendel, también conocida como la ley de la segregación independiente. Mientras que la primera ley de Mendel establece que los alelos se asignan aleatoriamente a la descendencia, la segunda ley se refiere a la independencia con la que se heredan los alelos para diferentes factores.

La ley de segregación independiente implica que los alelos para diferentes factores se heredan sin depender uno del otro. Esto significa que, si observamos la herencia de un factor, esta no afectará la herencia de otro factor. Por ejemplo, si cruzamos dos organismos heterocigotos para dos factores, la segunda ley de Mendel indica que la proporción de genotipos resultantes será la misma para cada factor individualmente.

Esta independencia nos permite calcular probabilidades de eventos relacionados con la herencia genética.

- Resolución:

El problema requiere calcular la probabilidad de ciertos genotipos en generaciones posteriores de una familia, asumiendo que la segunda ley de Mendel se cumple para los factores genéticos involucrados.

Para la solución del problema, se creó una función llamada independent_alleles1() que calcula la probabilidad de que al menos n individuos en una población de 2^K organismos tengan al menos un alelo específico en común, asumiendo una distribución de alelos independientes y una probabilidad constante de herencia.

El cálculo de la probabilidad se realiza mediante un bucle for que itera sobre todos los posibles números de individuos con el alelo específico, desde n hasta el tamaño total de la población. Para cada valor de i, se calcula la probabilidad utilizando la fórmula de combinación, multiplicando los factores siguientes:

- El coeficiente binomial, que representa el número de combinaciones de `i` individuos dentro de la población total de 2^K.
- La probabilidad de que i individuos tengan el alelo específico (0.25 elevado a la i, ya que hay una probabilidad de 0.25 para tener el alelo).
- La probabilidad de que los restantes (2^K - i) individuos no tengan el alelo (0.75 elevado a (2^K - i)).

Estos factores se multiplican y suman para obtener la probabilidad total de que al menos n individuos tengan el alelo específico.

Finalmente, la función devuelve la probabilidad calculada, redondeada al número especificado de decimales.

https://rosalind.info/problems/lia/

Finding a Protein Motif

- Descripción:

Se describe la estructura y función de las proteínas, centrándose en los dominios proteicos y los motivos. Los dominios son intervalos de aminoácidos que pueden evolucionar y funcionar de manera independiente, y cada uno suele corresponder a una función específica de la proteína. Los motivos son componentes esenciales de los dominios y se representan mediante un código de abreviatura. La identificación de proteínas y sus motivos se realiza en bases de datos como UniProt, que proporciona información detallada sobre la estructura de las proteínas y sus funciones.

- Resolución:

El problema plantea la identificación de proteínas que poseen un motivo específico de glicosilación N. Se proporcionan identificadores de acceso de UniProt y se requiere devolver los ID de acceso de las proteínas que contienen el motivo, junto con las ubicaciones en las que se encuentra el motivo en la secuencia de la proteína.

Primero, define una función obtener_Seq(protein_id) que recupera la secuencia de proteínas correspondiente a un identificador dado de UniProt. Utiliza la API REST de UniProt para recuperar los datos en formato FASTA.

Luego, define una expresión regular motif_regex que representa el motivo a buscar en las secuencias de proteínas. El motivo está definido como una secuencia que

comienza con "N", seguida de cualquier aminoácido excepto "P", luego "S" o "T", y finalmente cualquier aminoácido excepto "P".

Seguidamente, se implementa la función motif_matches_in_string(s, pattern) busca todas las coincidencias del motivo en una cadena de proteínas dada utilizando la expresión regular proporcionada. Devuelve una lista de posiciones donde se encuentran estas coincidencias en la cadena de proteínas.

Después, se realiza un bucle sobre una lista de proteínas (PROTEINS), y para cada proteína se obtiene su secuencia utilizando la función obtener_Seq(). Luego, se buscan las posiciones del motivo en la secuencia de proteínas utilizando la función motif_matches_in_string(). Si se encuentran coincidencias, se registra el nombre de la proteína y las posiciones de las coincidencias en una cadena de salida output.

Finalmente, la cadena output contiene los nombres de las proteínas y las posiciones donde se encontraron coincidencias del motivo.

https://rosalind.info/problems/mprt/

Inferring a mRNA from Protein

- Descripción:

Se comenta sobre los desafíos de revertir la traducción de proteínas a secuencias de ARN mensajero (mRNA). Aunque una cadena de ARN puede traducirse en una proteína única, revertir el proceso genera una gran cantidad de posibles cadenas de ARN a partir de una sola cadena de proteína debido a que la mayoría de los aminoácidos corresponden a múltiples codones de ARN.

- Resolución:

El problema requiere calcular el número de cadenas de ARN que podrían haberse traducido para producir la proteína dada, considerando la aritmética modular y evitando problemas de almacenamiento de números grandes. Se debe devolver este número total de cadenas de ARN módulo 1,000,000.

Para ello, se desarrolla una función mrna_from_protein() que cuenta la cantidad de codones de ARN que codifican cada aminoácido. La función utiliza un diccionario rna_codons que asigna codones de ARN a aminoácidos correspondientes. Itera sobre este diccionario para contar cuántos codones corresponden a cada aminoácido.

Luego, se calcula el número total de secuencias de ARNm que codifican la proteína dada. Itera sobre cada aminoácido en la proteína y multiplica el número de codones que codifican ese aminoácido por el contador acumulado. Este contador acumulado

se inicializa en 1 y se multiplica por la cantidad de codones para cada aminoácido encontrado en la proteína.

Finalmente, se multiplica este contador acumulado por el número de codones que codifican el aminoácido de parada (indicado como Stop) y toma el resultado módulo 1,000,000. Esto se hace para evitar que los números crezcan demasiado grandes. El resultado final representa el número de secuencias de ARNm que pueden producir la proteína dada, teniendo en cuenta las posibles variantes de codificación debido a la degeneración del código genético y se devuelve como resultado.

https://rosalind.info/problems/mrna/

Open Reading Frames

- Descripción:

Se describe sobre la traducción directa de una cadena de ADN en una cadena de proteínas, sin pasar por la etapa de ARN mensajero. Se señalan tres complicaciones principales: la presencia de ADN no codificante, la posibilidad de comenzar la traducción en cualquier posición de la cadena de ADN y la existencia de seis marcos de lectura posibles debido a las dos hebras del ADN y sus reversos complementarios.

- Resolución:

El problema requiere encontrar todas las posibles cadenas de proteínas candidatas que puedan ser traducidas a partir de marcos de lectura abiertos (ORFs) en la cadena de ADN dada. Un ORF comienza con un codón de inicio y termina con un codón de parada, sin contener otros codones de parada en el medio.

Primeramente, se implemento una función llamada open_reading_frames1() que encuentra y traduce marcos de lectura abiertos (ORFs) en una secuencia de ADN dada en su respectiva secuencia de proteínas.

La función define una función interna llamada translation() que traduce una secuencia de ARN a su correspondiente secuencia de aminoácidos. Itera sobre la secuencia de ARN en pasos de tres nucleótidos (codones) y traduce cada codón a un aminoácido utilizando un diccionario de codones de ARN proporcionado. Si encuentra un codón de parada, interrumpe la traducción y devuelve la secuencia de aminoácidos traducida hasta ese punto.

Luego, la función principal transcribe la secuencia de ADN a ARN utilizando el método transcribe() y la transcribe su complemento inverso utilizando el método reverse_complement() de la biblioteca BioPython.

Después, busca ORFs en ambas secuencias de ARN transcritas, comenzando desde cada uno de los tres marcos de lectura posibles. Para cada marco de lectura, busca el codón de inicio "AUG" y, a partir de ese punto, traduce la secuencia de ARN utilizando la función translation(). Si encuentra un ORF válido (es decir, una secuencia de aminoácidos que no contiene un codón de parada), lo agrega a una lista llamada ORFS.

Finalmente, la función devuelve una cadena que contiene todos los ORFs encontrados, evita duplicados utilizando set(), y los concatena con saltos de línea.

Este resultado proporciona una lista de todas las posibles secuencias de proteínas codificadas por los ORFs encontrados en la secuencia de ADN dada.

https://rosalind.info/problems/orf/

Enumerating Gene Orders

- Descripción:

Se explica cómo las mutaciones puntuales son capaces de generar cambios en poblaciones de organismos de la misma especie, pero carecen del poder para crear y diferenciar especies enteras. Esta tarea más ardua queda en manos de mutaciones más grandes llamadas reorganizaciones genómicas, las cuales mueven bloques enormes de ADN. Las reorganizaciones causan cambios genómicos importantes y la mayoría de ellas son fatales o dañinas para la célula mutada y sus descendientes. Debido a su rareza, las reorganizaciones que afectan la evolución de las especies son poco comunes.

Para comparar dos genomas de especies cercanamente relacionadas, los investigadores primero identifican intervalos de ADN similares de las especies, llamados bloques de sintenia, que han sido creados por reorganizaciones a lo largo del tiempo. Estos bloques se han movido a través de los genomas de las dos especies, a menudo separándose en diferentes cromosomas.

- Resolución:

Se requiere calcular el número total de permutaciones de longitud N, seguido de una lista de todas esas permutaciones, para un valor dado de N.

Se desarrollo una función llamada generate_permutations(), que genera todas las permutaciones posibles de números del 1 al N.

El proceso principal de generación de permutaciones se lleva a cabo dentro de una función interna llamada backtrack(). Esta función utiliza la técnica de backtracking para generar todas las permutaciones posibles. Comienza con una lista vacía y, recursivamente, agrega elementos a la lista y vuelve a llamar a la función hasta que

se alcanza una permutación completa (cuando la longitud de la lista es igual a N). Luego, agrega la permutación completa a la lista permutations.

Por otro lado, la función generate_permutations() también incluye una condición para limitar la generación de permutaciones si N es menor o igual a 7, lo que sugiere que el algoritmo puede ser ineficiente para valores grandes.

Después de generar todas las permutaciones, el código crea una salida de texto que contiene el número total de permutaciones generadas seguido de cada permutación en una línea separada, donde los números de cada permutación están separados por espacios.

https://rosalind.info/problems/perm/

Calculating Protein Mass

- Descripción:

Se comenta sobre el proceso de traducción de ARN en una cadena de aminoácidos para la construcción de una proteína. Durante este proceso, los aminoácidos se unen formando un enlace peptídico, lo que libera una molécula de agua. Después de que una serie de aminoácidos se han unido en un polipéptido, cada par de aminoácidos adyacentes ha perdido una molécula de agua. Los aminoácidos de los extremos no pierden esta molécula y tienen un enlace peptídico especial. La masa de una proteína es la suma de las masas de todos sus aminoácidos, más la masa de una molécula de agua.

Para calcular la masa de una proteína, se utilizan dos formas estándar: la masa monoistópica y la masa promedio. La masa monoistópica se calcula utilizando el isótopo principal de cada átomo en el aminoácido, mientras que la masa promedio se calcula utilizando el promedio de los isótopos de cada átomo en la molécula.

En el contexto de la espectrometría de masas, la masa monoistópica se usa con más frecuencia que la masa promedio. En este campo, se asume que todas las masas de los aminoácidos son monoistópicas a menos que se indique lo contrario.

- Resolución:

El problema plantea calcular el peso total de una cadena de proteína, utilizando la tabla de masas monoistópicas de aminoácidos.

Para la resolución del problema, se creó una función llamada protein_mass() que calcula la masa de una proteína a partir de la suma de las masas de sus aminoácidos.

Esta función utiliza una tabla de masas monoisotópicas predefinida llamada MONOISOTOPIC_MASS_TABLE. La función toma tres parámetros: protein (la secuencia de aminoácidos de la proteína), mass_table (una tabla que mapea los aminoácidos a sus masas) y decimals (el número de decimales a redondear en el resultado). Itera sobre cada aminoácido en la proteína, suma sus masas de acuerdo con la tabla y devuelve el total, redondeado según lo especificado.

https://rosalind.info/problems/prtm/

Longest Increasing Subsequence

- Descripción:

Se discute la comparación del orden de genes en cromosomas tomados de dos especies diferentes y modificados por reorganizaciones a lo largo de la evolución.

Una forma simple de comparar genes de dos cromosomas es buscar la mayor colección de genes del mismo orden en ambos. Para hacerlo, se utiliza la idea de permutaciones. Si dos cromosomas comparten n genes y etiquetamos los genes de un cromosoma con los números del 1 al n en el orden en que aparecen, el segundo cromosoma estará dado por una permutación de estos genes numerados. Para encontrar el mayor número de genes que aparecen en el mismo orden, solo necesitamos encontrar la mayor colección de elementos crecientes en la permutación.

- Resolución:

El problema consiste en encontrar la subsecuencia más larga creciente y la subsecuencia más larga decreciente de una permutación dada.

Se proporciona un número entero positivo N≤10000 seguido de una permutación π de longitud N y, de igual manera, se solicita devolver la subsecuencia creciente más larga de π seguida de la subsecuencia decreciente más larga de π.

Para encontrar la solucion, se implementaron dos funciones: longest_increasing_subsequence() y longest_decreasing_subsequence(). Ambas funciones toman una lista de números como entrada y devuelven la subsecuencia más larga creciente o decreciente de esa lista, respectivamente.

Para encontrar la subsecuencia más larga creciente, el código utiliza el algoritmo de programación dinámica. Inicialmente, crea una matriz m con todos los elementos iguales a 1. Luego, recorre la lista de entrada y compara cada elemento con los elementos anteriores. Si el elemento actual es mayor que un elemento anterior y su longitud aumenta, se actualiza la matriz m. Finalmente, encuentra la

longitud máxima de la subsecuencia creciente y luego retrocede para obtener los elementos de esa subsecuencia.

El proceso es similar para la subsecuencia decreciente, excepto que se verifica si el elemento actual es menor que los elementos anteriores.

El resultado final es una cadena que contiene la subsecuencia más larga creciente seguida por la subsecuencia más larga decreciente, ambas separadas por un salto de línea.

https://rosalind.info/problems/revp/

Enumerating Oriented Gene Orderings

- Descripción:

Se define el concepto de bloques de sintenia, que son regiones similares en el genoma de dos especies diferentes que han sido movidas y volteadas por reordenamientos. Para modelar cromosomas utilizando bloques de sintenia de manera más precisa, se les asigna una orientación para indicar la cadena en la que se encuentran.

- Resolución:

El problema consiste en determinar el número total de permutaciones firmadas de longitud N, seguido de una lista que las representa.

Una permutación firmada es aquella en la que cada número puede ser positivo o negativo.

Para abordar esta problemática, se diseñó la función generate_signed_permutations(). Esta función incluye un bucle condicional que verifica si N es par o impar. En caso de ser par, genera todas las permutaciones posibles sin restricciones adicionales. Sin embargo, si N es impar, se generan dos conjuntos de permutaciones: uno con todos los números del 1 al N y otro con el negativo de N incluido.

Además, se implementa una función auxiliar interna denominada helper(), la cual de manera recursiva genera todas las permutaciones firmadas posibles. Esta función utiliza un enfoque de retroceso (backtracking), iterando sobre los números restantes y generando las permutaciones posibles a partir de ellos.

El resultado final es una lista que contiene todas las permutaciones generadas, seguida por un recuento del total de permutaciones. Cada permutación se presenta en una línea separada por espacios, lo que constituye el dato de salida del proceso.

https://rosalind.info/problems/sign/