UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES
ACATLÁN

Elementos Matemáticos para la Programación Competitiva

TESINA

QUE PARA OBTENER EL TÍTULO DE

Lic. en Matemáticas Aplicadas y Computación

PRESENTA:

Saraí Ramírez Gómez

Asesor: Dr. Jorge Vasconcelos Santillán

Santa Cruz Acatlán, Estado de México, Enero 2020

Agradecimientos

A mis padres, quienes me han apoyado en cada paso de mi existencia de incontables maneras.

A Moroni, que ha aportado tantas genialidades a esta obra, a mi formación y a mi vida.

A mi asesor, el doctor Jorge Vasconcelos, por su guía, su apoyo y su paciencia, como profesor y como persona.

Al profesor Juan José Parres Córdova, que ha sido la inspiración de muchos de mis logros, esta obra incluida.

Al Departamento de Servicios de Cómputo de la FES Acatlán y a Araceli Pérez Palma, por todo el apoyo y la disposición para llevar a cabo este proyecto.

A mis brillantes y admirables amigos (a quienes me resulta complicado enlistar), que han sido un ejemplo a seguir como profesionistas y como seres humanos.

Y a todos ustedes, integrantes de GUAPA, que día con día se esfuerzan por mejorar, y que brindan su tiempo, paciencia y conocimiento a la formación de otros estudiantes.

Tabla de Contenido

Notación Empleada

Símbolo	Significado
$\mathbb{N}$	Números naturales
$\mathbb{Z}$	Números enteros
$\mathbb{R}$	Números reales
$a < b$	a es menor que b
$a > b$	a es mayor que b
$a \leq b$	a es menor o igual que b
$a \geq b$	a es mayor o igual que b
$a \in A$	a pertenece a A
$O(k)$	Complejidad de un algoritmo
$[a, b]$	Intervalo cerrado de a a b
(a, b)	Intervalo abierto de a a b

Símbolo	Significado
$< a, b, c >$	Arreglo compuesto por los elementos a, b y c
$\lceil k \rceil$	Función techo de k
$\lfloor k \rfloor$	Función piso de k
$\|A\|$	Cardinalidad del conjunto A
$k!$	Factorial de k
$\sum$	Suma
$\prod$	Producto
$\binom{n}{k}$	Coeficiente binomial de n en k
$\binom{n}{k_1,\, k_2}$	Coeficiente multinomial
$\overline{A}$	Complemento del conjunto A
$A \cup B$	Unión de los conjuntos A y B
$A \cap B$	Intersección de los conjuntos A y B
$[n]$	Conjunto de los primeros n números naturales
D_k	Subfactorial de k
$a \mid b$	a divide a b
$a \equiv b \ (mod\ n)$	a es congruente con b, módulo n
$mcd(a, b)$	Máximo común divisor de a y b

Tabla de Contenido

Símbolo	Significado
$mcm[a, b]$	Mínimo común múltiplo de a y b
$w(e)$ / $w(u, v)$	Peso de la arista e / Peso de la arista u, v
$N(v)$	Conjunto de vecinos del vértice v
$d(v)$	Grado del vértice v

Introducción

En esencia, la programación competitiva es un deporte mental cuyo objetivo es resolver eficientemente, y tan rápido como sea posible, problemas de computación, esto proponiendo un algoritmo para solucionar cierta situación planteada. Por lo general, estos concursos se realizan en equipos de tres personas y se cuenta con únicamente un equipo de cómputo (por lo que el trabajo en equipo posee un papel realmente importante dentro del entrenamiento de cualquier concursante), aunque existen eventos que no se alinean a estas características.

En la FES Acatlán existe el proyecto del Grupo Universitario de Algoritmia y Programación Avanzada (GUAPA), conformado únicamente por y para estudiantes, cuyo objetivo es la participación de los alumnos de la institución en diversos concursos de programación, logrando, por supuesto, cada vez mejores resultados. Éste es un proyecto relativamente nuevo, por lo que, desafortunadamente, al momento no existe un temario a seguir que permita homogeneizar el conocimiento de los participantes. Pensando en ello es que surge la idea de escribir este material, en el que se concentraran los temas necesarios para el entrenamiento autodidacta de toda persona interesada en el área.

El presente trabajo pretende presentar con profundidad y de un modo didáctico diversos temas matemáticos necesarios para la solución eficiente de una variedad de problemas en programación competitiva, con lo que, su principal objetivo es servir como guía a alumnos involucrados en esta área; esta obra podrá fungir como una herramienta que los instruirá desde conceptos básicos y, de forma paulatina, los adentrará a resultados más complejos. De manera paralela, este trabajo puede ser

de utilidad también para todos aquellos estudiantes de la carrera de Matemáticas Aplicadas y Computación y áreas afines en las materias de combinatoria, teoría de números, teoría de grafos y programación.

La mayoría de los concursos de programación siguen un formato muy similar: los participantes reciben un conjunto de problemas con requerimientos claramente definidos, y deben idear un algoritmo solución para cada uno de ellos. La eficiencia de las soluciones se evalúa a través de las salidas producidas por los mencionados algoritmos haciendo uso de un juez automático, mismo que arrojará alguno de los siguientes veredictos:

- ***Accepted.*** El problema se resolvió correctamente.
- ***Wrong Answer.*** El algoritmo propuesto es incorrecto y sus salidas no coinciden con las del juez.
- ***Time Limit.*** Cuando un problema se diseña, se estipula el tamaño de los datos a usar, y con ello se asigna un tiempo permitido para su ejecución [1]. Si el algoritmo propuesto excede dicho tiempo, aparecerá este veredicto.
  Hay que aclarar que este resultado no implica que las salidas del programa propuesto no sean iguales a las de la solución (es decir, que el programa sea erróneo), sin embargo, el algoritmo no es lo suficientemente veloz.
  Nos manejamos bajo la premisa de que una computadora promedio resuelve 10^8 operaciones por segundo, como se confirma en [2].
- ***Runtime Error.*** El programa no terminó de ejecutarse. Las causas pueden ser diversas, como una división entre 0 y el mal uso de memoria.
- ***Partially Accepted.*** Este veredicto únicamente es considerado por algunos jueces (el Juez GUAPA [3], de la FES Acatlán, entre ellos), y permite la aceptación de algoritmos que son correctos, pero no eficientes; esta herramienta es útil para aquellos participantes que están en los inicios de su entrenamiento.

Prepararse para un concurso de programación competitiva implica el estudio integral de diversas materias, y en esta preparación es tan importante el saber programar como lo es el tener sólidos y profundos conocimientos matemáticos. Imagina, por ejemplo, que te enfrentas a un problema que requiere que obtengas la suma de los primeros n números naturales; si decides delegar todo el trabajo de la adición a la

computadora, programarás un ciclo de repetición de tamaño n, que vaya sumando, uno a uno, los enteros que necesitas, procedimiento de complejidad $O(n)$, lo que implica que el tiempo de ejecución aumentará conforme aumente el tamaño de n; por otro lado, si tu conocimiento matemático abarca la llamada fórmula de Gauss, sabrás que esa misma suma se resume a la expresión $\frac{n(n+1)}{2}$, y resolverla tiene una complejidad constante, lo que no cambiará aún si n se vuelve extremadamente grande. Este sólo es un sencillo ejemplo de cómo podemos utilizar resultados matemáticos para agilizar un proceso.

En esta obra hablaremos de Análisis Combinatorio, Teoría de Números y Teoría de Grafos, mismos que se se unirán con algunas técnicas avanzadas de resolución de problemas, en donde se evidenciará que la teoría matemática vista en los mencionados capítulos carga un papel importante en la programación de diversos procesos. Además, en los Anexos encontrarás una breve introducción al Análisis de Algoritmos, las funciones más comunes empleadas en la obra y un muestrario de problemas que han sido utilizados en diferentes concursos.

Aunque el desarrollo de las habilidades para el trabajo en equipo y la preparación en programación y matemáticas son aspectos claramente importantes, la característica más valorada de un estudiante que pretende dedicar su tiempo a programar competitivamente es su disposición para ser autodidacta, pues al final, el único límite de un competidor será la frontera de donde termina su deseo de aprender. Este material, pues, pretende servir como apoyo a todos aquellos estudiantes que, pese a su deseo por aprender, no tienen al alcance la teoría matemática amalgamada con su aplicación para la creación de algoritmos eficientes; dicho sea de paso, ahondar en los temas que aquí se presentan es de suma utilidad en entrevistas de trabajo hechas por grandes compañías de software internacionales. Entendiendo que el área puede ser complicada, se ha escrito esta obra en un estilo lo más amigable posible, buscando que el lector encuentre en ella una conversación explicativa de la teoría, acompañada de numerosos ejemplos que muestren desde diferentes ángulos la aplicación de lo visto en cada sección, con lo que se pretende que, más allá de memorizar fórmulas y soluciones, cada estudiante sea capaz de comprenderlas y de crear las suyas propias a partir del conocimiento adquirido y del desarrollo de su razonamiento, y es que, ciertamente, este escrito es ideal para ello, pues mucho del conocimiento que aquí se aborda surge de la resolución de diversos problemas, no únicamente de la consulta de fuentes.

Como seguramente habrás escuchado muchas veces ya, lo importante de la programación no es el lenguaje en el que se codifica un algoritmo, sino la lógica detrás de él, por lo que puedes implementar los algoritmos que aquí encontrarás en el lenguaje de tu preferencia. Este material desarrolla sus algoritmos en C++11[1]; los segmentos de programa presentados no contienen una función principal o *main* con la finalidad de no extender el ejemplo y perder la atención del objeto de estudio, por lo que su ejecución requiere su invocación dentro de la función principal, o bien, dentro de otra función. La razón de la preferencia por C++11 es que su librería de plantillas estándar vuelve más amigable el uso de estructuras de datos y funciones relacionadas a ellas, por lo que nos brinda la oportunidad de enfocar nuestra atención en la resolución lógica de los problemas, en lugar de invertir tiempo en los detalles de sintaxis, y es que ciertamente, esta obra no está enfocada al aprendizaje del lenguaje (el cual puede realizarse en [5]), sino a su uso como una herramienta del conocimiento matemático.

Dicho lo anterior, estás listo para lidiar con las siguientes páginas.

[1] Bajo las reglas [4] de la mayoría de los concursos de programación competitiva, es necesario asegurar la solución en C y en C++.

1

Análisis Combinatorio

Frecuentemente, en programación competitiva se presentan problemas en los que requerimos contar los elementos de un conjunto de datos relativamente grande. Sin embargo, contar elemento por elemento no suele ser buena idea, ya que el procedimiento se vuelve lento conforme aumenta el tamaño del conjunto.

En este capítulo estudiaremos cómo aprovechar el análisis combinatorio, o combinatoria, para reemplazar varias líneas de código por fórmulas sencillas, que reduzcan la complejidad de la solución que propongamos.

Varios de los resultados que veremos aquí te podrán parecer muy intuitivos y poco sofisticados, sin embargo, su teorización será necesaria para su aprovechamiento real dentro de problemas de alto nivel.

1.1. Conceptos Básicos

1.1.1. Principio Fundamental de Conteo

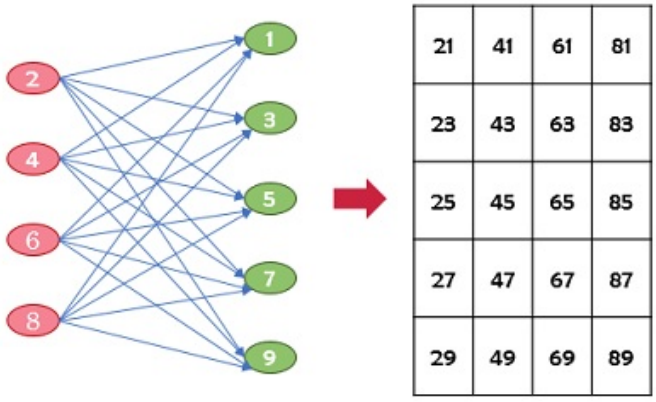
Este principio se compone de dos fundamentos:

Principio Aditivo. Si deseamos realizar cierta tarea, misma que puede llevarse a cabo de formas múltiples, donde la primera forma puede ocurrir de m maneras, y la segunda de n formas, entonces una operación o la otra puede ejecutarse de $m + n$ modos.

Principio Multiplicativo. Si podemos realizar cierta tarea de n_1 formas distintas y para cada una de ellas existe una segunda tarea que se puede realizar de n_2 maneras diferentes, entonces ambas tareas se pueden realizar de $n_1 \cdot n_2$ formas.

Ejemplo 1. ¿Cuántos números de dos dígitos existen de forma que el dígito de las decenas sea par y el dígito de las unidades sea impar?

Solución. Para el dígito de las decenas existen 4 opciones viables: 2, 4, 6 y 8 (pues el 0 no sería significativo en esa posición), mientras que para el dígito de las unidades tenemos cinco posibilidades: 1, 3, 5, 7 y 9. Por lo tanto, existen $4 \cdot 5 = 20$ números de la forma requerida.



Ejemplo 2. Al levantarte por la mañana te preguntas cómo te vestirás hoy. Digamos que eres una persona indecisa y quieres probar todas las formas posibles en que podrías combinar tu ropa. Supongamos que tus opciones son cuatro playeras, dos pantalones y tres pares de zapatos. ¿Cuántos cambios de ropa tendrás que hacer?

Solución. Por el principio fundamental de conteo, probar las combinaciones de las 4 playeras y los dos pantalones implica $4 \cdot 2 = 8$ cambios.

Luego, probar cada uno de esos cambios con los tres pares de zapatos resultaría en $8 \cdot 3 = 24$ cambios.

Ejemplo 3. Considerando que en el alfabeto latino moderno hay 26 letras, ¿cuántas cadenas de 5 letras es posible formar?

Solución. Para cada una de las letras de las cadenas tenemos las 26 opciones en el alfabeto. Por tanto, podemos formar $26 \cdot 26 \cdot 26 \cdot 26 \cdot 26 = 26^5 = 11,881,376$ cadenas.

Ejemplo 4. Imagina una matriz de $n \times n$. ¿Cuál será la complejidad de recorrerla completamente?

Solución. Independientemente del ciclo de repetición que elijas, deberás anidar un ciclo dentro de otro, de tal forma que uno recorra las filas y otro las columnas, y ya que ambos van de 0 a $n - 1$, podemos decir que para cada valor que tome el primer ciclo, se deberán recorrer los n valores del segundo, con lo que, por el principio fundamental de conteo, obtenemos una complejidad de $O(n^2)$.

Los ejemplos anteriores nos muestran de una manera sencilla cómo funciona el teorema fundamental de conteo; en los siguientes ejercicios generalizaremos las soluciones para cualquier número natural.

Ejemplo 5. ¿Cuántos números de n dígitos existen de forma que todos sus dígitos en posiciones pares sean pares y todos sus dígitos en las posiciones impares sean impares, donde n es un entero positivo par?

Nombraremos n al dato de entrada, y la cantidad de números de n dígitos que cumplan con las condiciones requeridas será el dato de salida.

Solución. Consideremos a la posición de las unidades como el último dígito.

Al igual que en el ejemplo 1, el problema se resuelve aplicando el principio fundamental de conteo, sólo que esta vez debemos considerar más

datos. Nuevamente, tomamos en cuenta que para el primer dígito existen únicamente 4 opciones viables, sin embargo, para los $n-1$ dígitos restantes, sin importar si están en una posición par o en una posición impar, hay 5 posibilidades. Por lo tanto, la fórmula que se debe programar para solucionar este problema es $4 \cdot 5^{n-1}$.

```
1  //Obtenemos la potencia requerida.
2  int potencia(int base, int pot)
3  {
4      int i, res = 1;
5      for (i = 0; i < pot; i = i + 1)
6          res = res * base;
7
8      return res;
9  }
10
11 //Programamos la fórmula a la que llegamos.
12 int solucion()
13 {
14     int n;
15     cin>>n;
16     return 4 * potencia(5, n - 1);
17 }
```

La primera función en el programa nos servirá para obtener cualquier potencia, en particular, 5^{n-1}, y en lo sucesivo haremos referencia a ella en repetidas ocasiones.

Ejemplo 6. Nuevamente, intentado vestirte, quieres probar todas las posibles formas en que podrías hacerlo. Imaginando que somos seres extraños que pueden usar muchas prendas distintas simultáneamente, tendrás como entrada un número natural n, que indicará la cantidad de prendas distintas que puedes usar al mismo tiempo; luego vendrán n números naturales, cada uno para indicar la cantidad de opciones que hay para la prenda i, para $i = 1, 2, \ldots, n$.

Como salida, buscamos todos los cambios de ropa que tendrás que hacer.

Solución. Llamemos $a_1, a_2, \ldots, a_n$ a los números que nos indican la cantidad de opciones para la i-ésima prenda. Por el principio fundamental de

conteo, existen $\prod_{i=1}^{n} a_i$ formas de hacerlo, es decir, el producto de todas las a_i's dadas.

```
int solucion()
{
    int i, n, respuesta = 1;
    cin>>n;
    int a[n + 1];
    for (i = 1; i < n + 1; i = i + 1)
        cin>>a[i];

    for (i = 1; i < n + 1; i = i + 1)
        respuesta = respuesta * a[i];

    return respuesta;
}
```

Nota que, en el fragmento de programa, almacenamos los n naturales en un arreglo, para luego multiplicar entre sí todos sus elementos.

Ejemplo 7. ¿Cuántas palabras (secuencias de letras) de n letras es posible formar con el alfabeto latino moderno?

Solución. Considerando que existen 26 letras en el alfabeto:

```
int solucion()
{
    int i, n, respuesta = 1;
    cin>>n;

    for (i = 0; i < n; i = i + 1)
        respuesta = respuesta * 26;

    return respuesta;
}
```

1.1.2. Permutaciones

Se le llama permutación a una biyección de un conjunto A sobre sí mismo, es decir, una permutación es una función biyectiva $f : A \to A$.

Dicho de una forma más simple, teniendo una colección de n objetos *distintos*, las permutaciones cuentan de cuántas formas es posible acomodarlos.

Veamos que para acomodar los n objetos de los que hablamos tenemos n lugares. En el primero de esos sitios podemos colocar cualquiera de los n objetos. Sin pérdida de generalidad, para el segundo lugar tendremos todos los objetos a excepción del que hemos acomodado ya en el primero, por lo que existen $n - 1$ opciones viables. Para el tercer lugar usamos el mismo razonamiento y vemos que aquí se podrán colocar todos los objetos salvo los que se utilizaron en el primer y el segundo lugar. Si aplicamos la misma lógica para el resto de los lugares, cuando lleguemos al $(n-1)$-ésimo lugar, para éste habrá únicamente dos opciones, y finalmente, para el n-ésimo lugar, habrá una, el único elemento que no ha sido utilizado. Así, aplicando el principio fundamental de conteo, existen $n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$ configuraciones distintas para los n, producto que puede ser escrito como $n!$; esta expresión se conoce como el *factorial de* n.

Ejemplo 8. ¿Cuántas cadenas es posible construir con las 26 letras del alfabeto latino de tal modo que todas aparezcan exactamente una vez?

> **Solución.** A diferencia del ejemplo 7, aquí tenemos la condición extra de no poder repetir caracteres en una cadena, lo que implica que sólo podremos "cambiar de lugar" las letras (y no repetirlas), es decir, permutarlas.
>
> En el primer lugar de la cadena podemos colocar cualquiera de las 26 letras. Luego, como no es posible repetir las letras que se usan, para el siguiente lugar se tienen 25 posibilidades. Siguiendo el mismo razonamiento hasta el último lugar, llegamos a que existen 26! cadenas distintas, internándonos así en el uso del factorial.

Ejemplo 9. ¿De cuántas formas es posible sentar a 5 individuos en 5 asientos contiguos?

Solución. En el primer asiento podemos colocar a cualquiera de las 5 personas. Ya que ésta ha sido sentada, para la segunda silla hay 4 posibilidades. Para la tercera silla aún hay 3 opciones, para la cuarta 2 y para la última únicamente 1. Por tanto, hay $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5! = 120$ formas de sentar a estas 5 personas.

Generalizando el ejemplo anterior para n personas y n asientos, estos individuos pueden ser sentados de $n!$ formas, lo que, programado, se puede obtener de la siguiente manera.

```
int factorial()
{
    int n;
    cin>>n;

    if (n == 0)
        return 1;

    return n * factorial(n - 1);
}
```

Acabamos de crear una función que, además de resolver este ejercicio, calcula el factorial de un número natural. Recuérdala bien, ya que más adelante haremos uso de ella.

Ejemplo 10. ¿Cuántas cadenas de k letras existen, para $k \leq 26$, si debe cumplirse que no se repita letra alguna en dichas cadenas?

Solución. Una forma de resolver el ejercicio es imitando la solución del problema 8, considerando la condición de que esta vez únicamente contamos con k lugares para colocar letras, donde k no necesariamente es igual a 26. Así, la solución de este ejercicio será $(26)(25)\cdots(26-k+2)(26-k+1)$.

```
int solucion()
{
    int i, k, respuesta = 1;
    cin>>k;

    for (i = 26; i > k; i = i - 1)
        respuesta = respuesta * i;
```

```
8
9       return respuesta;
10  }
```

Este ejemplo nos lleva a preguntarnos qué sucede cuando deseamos contar las permutaciones de un conjunto de tamaño n si sólo se tienen k lugares para ellos, con $k < n$.

En realidad, ya que resolvimos los ejercicios anteriores, llegar a la respuesta de esta pregunta será muy sencillo, pues haremos uso de un razonamiento muy similar: En el primero de los k lugares es posible colocar cualquiera de los n objetos, para el siguiente se pueden colocar $n-1$ y así sucesivamente hasta llegar al k-ésimo lugar, para el que habrá $n-(k-1)$ posibilidades. Una vez más, aplicando el principio fundamental de conteo, existen $n \cdot [n-1] \cdots [n-(k-2)] \cdot [n-(k-1)]$ formas de acomodar n objetos en k lugares.

Jugando un poco con la expresión que obtuvimos y bajo la consideración de que contamos con n objetos y k lugares, para $k < n$, podemos ver lo siguiente:

$$\begin{aligned} Permutaciones &= n \cdot [n-1] \cdots [n-(k-2)] \cdot [n-(k-1)] \\ &= n \cdot [n-1] \cdots [n-(k-2)] \cdot [n-(k-1)] \cdot \frac{(n-k)!}{(n-k)!} \\ &= \frac{n!}{(n-k)!} \end{aligned}$$

Así, lo que hicimos en el ejemplo 10 es equivalente a simplemente aplicar la fórmula $\frac{n!}{(n-k)!} = \frac{26!}{(26-k)!}$, algo que es más sencillo de programar una vez que hemos comprendido el funcionamiento de un factorial.

```
1  //Calculamos el factorial con la función de la que recientemente hablamos.
2  int factorial(int k)
3  {
```

```
4       if (k == 0)
5           return 1;
6
7       return k * factorial(k - 1);
8   }
9
10  int solucion()
11  {
12      int k;
13      cin>>k;
14      return factorial(26) / factorial(26 - k);
15  }
```

1.1.3. Combinaciones

El factor que diferencia a las permutaciones de las combinaciones es la importancia del orden de los objetos. Mientras que cuando contamos permutaciones queremos saber de cuántas formas es posible acomodar los objetos en juego, cuando contamos combinaciones buscamos obtener la cantidad de todos los posibles subconjuntos de tamaño k de un conjunto de cardinalidad n, con $k \leq n$, sin importar el orden que tomen los objetos dentro de los subconjuntos.

Para contar todos los subconjuntos de tamaño k de un conjunto de mayor o igual tamaño, digamos n, recurrimos a lo que se conoce como *coeficientes binomiales*. Estos se denotan como $\binom{n}{k}$, y se leen como *"n en k"*, y su significado matemático es el siguiente:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Por ejemplo, sea el conjunto $A = \{a, b, c\}$. Existen 6 $(= 3!)$ permutaciones de él mismo: *abc*, *acb*, *bac*, *bca*, *cab* y *cba*; sin embargo, si queremos contar las combinaciones de tamaño 3, sólo existe una: en la que están presentes todos los elementos de A; por otro lado, cuando contamos las permutaciones de tamaño 2 de A, existen $3(2) = 6$ acomodos: *ab*, *ba*, *ac*, *ca*, *bc* y *cb*, mientras que cuando contabilizamos las

combinaciones de tamaño 2, vemos que únicamente son 3: ab, bc y ca, y este número coincide con el resultado del coeficiente binomial $\binom{3}{2} = \frac{3!}{2!\cdot 1!} = \frac{6}{2} = 3$.

Ahora que conocemos los coeficientes binomiales y su significado veremos por qué estos cuentan las combinaciones de un conjunto (y es que, ciertamente, la relación no es evidente).

Sea el conjunto A cuya cardinalidad es n. Utilizando los objetos que pertenecen a A, queremos cubrir k espacios sin importarnos el orden, donde $k \leq n$, y contar las maneras en que esto se puede realizar. Dicho de otra forma, deseamos contabilizar las combinaciones de tamaño k. Ahora bien, con lo que estudiamos en la sección de *Permutaciones*, sabemos que existen $(n)(n-1)\cdots(n-k+1)$ permutaciones de tamaño k del conjunto A, sin embargo, para cada una de esas permutaciones se cumple que existe un conjunto de permutaciones que, en términos de combinaciones, son equivalentes (las permutaciones, por ejemplo, contarán a ab y ba como elementos distintos, pero en términos de combinaciones los mencionados elementos son equivalentes); para responder cuántas permutaciones son "equivalentes", hay que ver que éstas son conjuntos de cardinalidad k, por lo tanto, para cada conjunto distinto existen $k!$ permutaciones, es decir, por cada $k!$ permutaciones hay únicamente *una* combinación. Así, para contar las combinaciones, obtenemos lo siguiente la expresión:

$$\frac{n(n-1)\cdots(n-k+1)}{k!}$$

Para simplificarla, recurriremos al típico truco de multiplicar la expresión anterior por un 1 que nos convenga y jugaremos con un poco de álgebra.

$$\begin{aligned}\frac{n(n-1)\cdots(n-k+1)}{k!} &= \frac{n(n-1)\cdots(n-k+1)}{k!}\cdot\frac{(n-k)!}{(n-k)!}\\ &= \frac{[n(n-1)\cdots(n-k+1)][(n-k)(n-k-1)\cdots(2)(1)]}{(n-k)!k!}\\ &= \frac{n!}{k!(n-k)!} = \binom{n}{k}\end{aligned}$$

A partir de este momento, veremos varios ejemplos que encuentran una solución en el uso de coeficientes binomiales, sin embargo, debes tener en cuenta que éstas servirán únicamente cuando se trabaje con enteros menores o iguales que $\binom{66}{33}$, esto debido a que en C++ no existen tipos de datos que soporten un número mayor. Si este número es superado, habrá que hacer uso de técnicas más complejas que abordaremos en el capítulo de *Técnicas Avanzadas para la Resolución de Problemas*. Ten en cuenta que estas técnicas no excluyen las bases matemáticas, por lo que, para comprenderlas y dominarlas, es necesario que tu comprensión sobre este capítulo sea significativamente bueno.

Ejemplo 11. En un salón de clases con 15 personas se pretenden hacer equipos de 5 integrantes. ¿Cuántos equipos distintos es posible formar?

> **Solución.** Puedes ver que para resolver este problema el orden en que las personas se colocan dentro de cada equipo no es relevante (y esto es precisamente lo que dicta que estamos tratando con combinaciones y no con permutaciones).
>
> Entonces, ahora que comprendimos el funcionamiento de un coeficiente binomial para contar combinaciones, hagamos uso de la fórmula; en este caso particular, $n = 15$ y $k = 5$, pues tenemos un total de 15 personas de donde queremos contar todos los posibles subconjuntos de tamaño 5.
>
> $$\binom{n}{k} = \binom{15}{5} = \frac{15!}{5! \cdot (15-5)!} = \frac{15!}{5! \cdot 10!} = 3003$$

Ejemplo 12. ¿Cuántos equipos distintos con 5 personas pueden formarse si en cada equipo debe haber exactamente 3 hombres y 2 mujeres, considerando que el total de hombres en el salón es h y el total de mujeres es m?

Considera a h y a m como las entradas de tu programa y al total de equipos como la salida.

> Existen $\binom{h}{3}$ tríos distintos de hombres y hay $\binom{m}{2}$ pares distintos de mujeres.

Por el principio fundamental de conteo, se tienen $\binom{h}{3}\binom{m}{2}$ formas de agrupar, tanto a hombres como mujeres, de forma que se cumplan las condiciones requeridas, pues para cada trío distinto de hombres se deben considerar todos los posibles pares de mujeres.

```
//Creamos la función para obtener un coeficiente binomial.
int binomial(int n, int k)
{
    return factorial(n) / factorial(k) / factorial(n - k);
}

int solucion()
{
    int h, m;
    cin>>h>>m;
    return binomial(h, 3) * binomial(m, 2);
}
```

En este programa creamos la función *binomial*, con la que obtuvimos el valor de un coeficiente binomial dado, y será de gran importancia en varios puntos de esta obra.

Ejemplo 13. ¿Cuántas manos distintas de k cartas existen del mazo de una baraja española (que consta de 40 cartas)?

Solución. Básicamente, lo que queremos hacer es contar todos los subconjuntos de tamaño k de un conjunto de tamaño 40. Esto nos lleva a la expresión $\binom{40}{k}$.

```
int solucion()
{
    int k;
    cin>>k;
    return binomial(40, k);
}
```

Ejemplo 14. Queremos crear un programa que genere contraseñas basadas en alguna cadena dada C, todas del mismo tamaño, y claro, todas distintas entre sí; C

estará compuesta únicamente por letras mayúsculas del alfabeto latino moderno, no tendrá repeticiones de letras y tendrá una longitud t; no obstante, antes de crear dichas contraseñas necesitamos saber si existirán las suficientes para cierto número de individuos, así que primero debemos desarrollar un programa que cuente las cadenas que es posible formar.

En resumen, dada la cadena C, de longitud t, vamos a crear un programa que determine cuántas cadenas de tamaño k, para $k < t$, es posible formar utilizando únicamente las letras de C.

Por ejemplo, dada la cadena "HOLA", cuyo tamaño es 4, se tienen 12 subcadenas tamaño 2: "HO", "HL", "HA", "OH", "OL", "OA", "LH", "LO", "LA", "AH", "AO" y "AL".

Solución. Aprovecharemos este ejemplo para explicar cómo funcionan este tipo de permutaciones desde otro enfoque; ya caminamos de permutaciones a combinaciones, y en este ejemplo iremos en dirección opuesta.

Ya que $k < t$, lo primero será contar todos los subconjuntos de cardinalidad k del conjunto de las t letras de C, esto es $\binom{t}{k}$. Luego, *para cada subconjunto* contamos todas las posibles permutaciones de sus elementos, lo que podemos traducir en la expresión $k!$. Así, contaremos todas las posibles cadenas de tamaño k con el producto:

$$\binom{t}{k} k!$$

Desarrollando la expresión anterior:

$$\binom{t}{k} k! = \frac{t!}{\cancel{k!}(t-k)!} \cdot \cancel{k!}$$

Una vez más, obtenemos la expresión $\frac{t!}{(t-k)!}$. Codificándola:

```
1  int solucion()
2  {
3      int k;
4      cin>>k;
```

```
5       return factorial(t) / factorial(t - k);
6   }
```

El ejemplo anterior es una clara muestra de que es posible resolver un problema de múltiples maneras, y la mejor será aquélla en la que tú logres abstraer la información (siempre y cuando la complejidad sea lo suficientemente buena).

Ejemplo 15. Se tiene una cuadrícula de $m \times n$. ¿Cuántos paralelogramos se pueden formar con las líneas que conforman dicha cuadrícula?

Solución. La construcción de un paralelogramo requiere la elección de dos pares de lados, obviamente paralelos; en este caso, además, requeriremos que los mencionados pares de lados sean perpendiculares entre sí (pues estamos partiendo de una cuadrícula). Dicha elección será la clave para resolver el problema.

Dado que la base de la cuadrícula tiene m cuadros de base, existen $m+1$ líneas verticales, de las cuales debemos escoger dos. Entonces, ¿cuántos pares de líneas verticales se pueden formar en un conjunto de cardinalidad $n+1$? La respuesta es $\binom{n+1}{2}$.

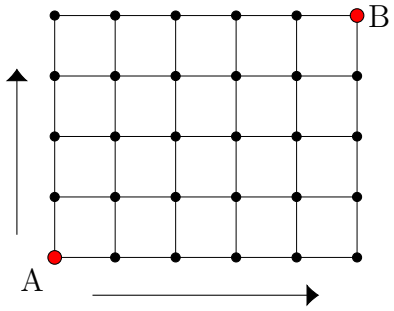
Análogamente, ¿cuántos pares de líneas horizontales se pueden formar de un conjunto de tamaño $m+1$? $\binom{m+1}{2}$.

Ya que para cada variación de una de las dimensiones se deben considerar todas las variaciones de la otra, aplicamos el principio fundamental del conteo, obteniendo el producto de las dos expresiones que acabamos de obtener:

$$\binom{m+1}{2}\binom{n+1}{2}$$

```
1   int solucion()
2   {
3       int m, n;
4       cin>>m>>n;
5       return binomial(m + 1, 2) * binomial(n + 1, 2);
6   }
```

Ejemplo 16. Sea una cuadrícula de $m \times n$, donde m y n serán las entradas del problema. Llamemos A al punto extremo inferior izquierdo y B al punto extremo superior derecho. Haciendo uso de únicamente movimientos hacia la derecha y hacia arriba se quiere llegar del punto A al punto B. ¿De cuántas formas puede realizarse esa tarea?



Solución. Podemos considerar cada secuencia de movimientos como una cadena que represente una sucesión de decisiones: si avanzamos hacia arriba o hacia la derecha. Independientemente de cuáles sean esas decisiones, recorreremos siempre $m + n$ unidades dentro de la cuadrícula, y siempre serán m unidades hacia la derecha y n hacia arriba. La solución pues, está en contar todas las posibles configuraciones de las decisiones que podemos tomar.

Denotemos como h a los pasos horizontales y como v a los pasos verticales. Podemos ver cada sucesión de decisiones como una cadena que debe ser compuesta por exactamente m h's y exactamente n v's. Por ejemplo, las cadenas $hhhhhvvvv$, $hvhvhvhvh$ y $vvvvhhhhh$ son algunas de las soluciones para la cuadrícula que se muestra arriba. Nota que si le damos lugar a las m decisiones h, todas las decisiones v tendrán un lugar en automático, pues serán todos los que restan. Luego, la solución se reduce a contar de cuántas formas se pueden elegir esos m lugares de entre las $m + n$ decisiones que se deben tomar en total, esto es $\binom{m+n}{m}$.

```
1  int solucion()
2  {
3      int m, n;
4      cin>>m>>n;
```

```
5       return binomial(m + n, m);
6   }
```

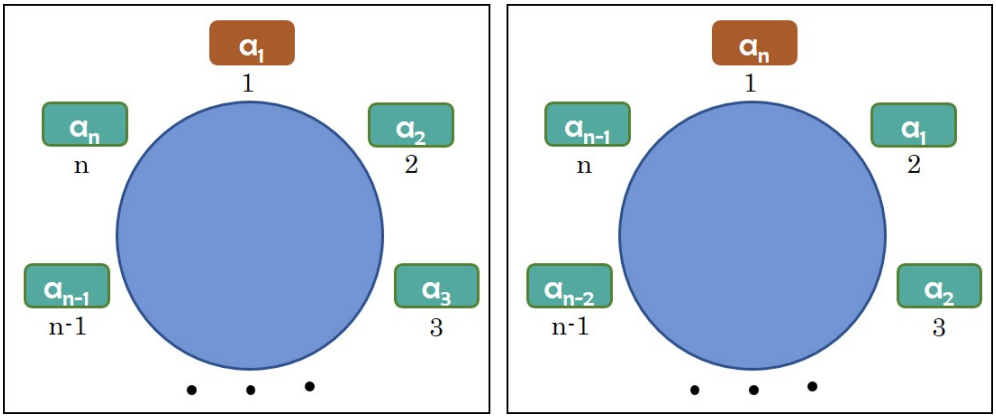
1.2. Permutaciones Circulares

Altos mandatarios de muchos países, digamos n, están invitados a una reunión muy importante en la que se tomarán decisiones que tendrán un alto impacto en el futuro del planeta. Como parte de la organización, tú estás a cargo de la distribución de sus lugares alrededor de una única mesa redonda, en donde todos se sentarán simultáneamente, y en la que existe la cantidad exacta de sillas para los mandatarios. Si quieres probar todos los posibles acomodos para así elegir el que luzca más conveniente, ¿cuántos acomodos distintos debes hacer, tomando en cuenta que todas las rotaciones de un acomodo se consideran iguales?

El inconveniente con este tipo de problemas es que claramente no podemos tratarlos como permutaciones lineales, pero podemos utilizar el conocimiento que hemos desarrollando anteriormente para comenzar a buscar una solución.

Enumeremos las sillas alrededor de la mesa de 1 a n dándole arbitrariamente a alguna de ellas la etiqueta del primer asiento, a su derecha estará el asiento número 2, posteriormente el 3 y así sucesivamente hasta asignarle a la silla a la izquierda del asiento 1 la etiqueta del n-ésimo. En la silla número 1 podemos colocar a n personas, en la segunda, $n-1$ y así sucesivamente, hasta que para la última silla sólo hay una opción, es decir, para darle algún lugar a todos se pueden hacer $n!$ posibles acomodos distintos; no obstante esta forma de contar considera como distintas a las rotaciones de una misma sucesión de personas, así que aún debemos pensar en algo que resuelva ese conflicto.

La siguientes imágenes muestran la enumeración de los asientos así como dos rotaciones de una misma sucesión de mandatarios, donde cada a_i representa a una persona distinta.



Para remediar esta situación debemos responder una pregunta: ¿Cuántas rotaciones existen para cada una de las sucesiones distintas de mandatarios?

Dado que disponemos de n lugares, es posible rotar cada sucesión de personas n veces, lo que significa que para cada sucesión distinta de mandatarios hay n rotaciones iguales, y éste es el ajuste que necesitamos hacer para contar los acomodos correctamente. Por tanto, obtenemos la expresión

$$\frac{n!}{n} = (n-1)!$$

Formalmente hablando, dados n objetos $a_1, a_2, \ldots a_n$ que queremos acomodar en n lugares alrededor de una circunferencia (que no puede ser rotada), existen $(n-1)!$ acomodos distintos.

Ejemplo 17. A una fiesta llegan n matrimonios, es decir, $2n$ personas, que deben sentarse alrededor de una gran mesa redonda. ¿De cuántas formas pueden hacerlo si todas las personas deben sentarse junto a su respectiva pareja?

Solución. Ya que no queremos desarticular a las parejas, vamos a considerar a cada matrimonio como una unidad. De esta forma, tendríamos que contar las permutaciones circulares de n elementos, esto es $(n-1)!$.

Ahora falta contar de cuántas formas puede permutarse internamente cada pareja. Con esto queremos decir que sólo hemos contado las formas de sentar a todas las parejas juntas, pero aún no consideramos el orden

que toman las personas como individuos. Para ello, usamos el hecho de que cada matrimonio está compuesto por dos personas, lo que implica que para cada pareja hay dos permutaciones.

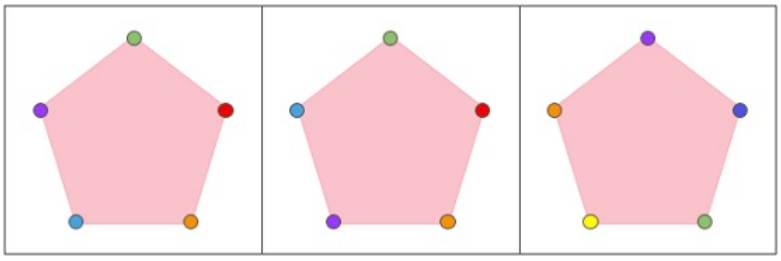
Finalmente, consideramos tanto las permutaciones de todas las parejas entre sí como las permutaciones internas, y obtenemos $2^n(n-1)!$.

Programar esta solución será rápido, pues ya hemos visto cómo calcular la potencia y el factorial de un número en los ejemplos 5 y 10, respectivamente.

```
int solucion()
{
    int n;
    cin>>n;
    return factorial(n - 1) * potencia(2, n);
}
```

Ejemplo 18. Dado un polígono de n lados, ¿de cuántas formas es posible colorear todos sus vértices con k colores, donde $k \geq n$, si cada color puede ser utilizado a lo sumo una vez?

Por ejemplo, si tenemos un pentágono y los 7 colores del arco iris, las siguientes imágenes nos muestran algunas de las coloraciones que podemos obtener.



Solución. Para el caso en el que $n = k$, dado que hay tantos colores como vértices, existen $(n-1)!$ coloraciones distintas.

Consideremos entonces el caso en el que $k > n$. Nota que hay más de

un subconjunto de tamaño n entre los k colores, con exactitud, hay $\binom{k}{n}$. Luego, para cada uno de esos subconjuntos hay $(n-1)!$ permutaciones circulares (pues cada uno se compone de exactamente n elementos). Por lo tanto, tenemos $\binom{k}{n}(n-1)!$ formas de pintar el polígono.

Probablemente para este momento estés pensando en hacer un programa en el que diferencies los casos en los que $k = n$ y $k > n$, sin embargo, no es necesario, ya que cuando $k = n$, $\binom{k}{n} = 1$, así que el producto de $\binom{k}{n}(n-1)!$ no es distinto de únicamente $(n-1)!$.

```
1  int solucion()
2  {
3      int n, k;
4      cin>>n>>k;
5      return binomial(k, n) * factorial(n - 1);
6  }
```

1.3. Multinomial: Permutaciones con Repeticiones

Anteriormente estudiamos cómo contar las permutaciones de un conjunto compuesto de objetos distintos, sin embargo, debemos preguntarnos cómo podemos contar las permutaciones de un conjunto en el que se presentan repeticiones entre sus elementos. Por ejemplo, para un conjunto de 3 objetos distintos, tenemos 6! permutaciones, pero si tenemos el conjunto $\{a, a, b\}$, únicamente tendremos las tres permutaciones *aab*, *aba* y *baa*.

Sea A un grupo con n elementos no necesariamente distintos, y sean $a_1, a_2, \ldots, a_k$ todos los elementos distintos en A. Nombramos como $r_1, r_2, \ldots, r_k$ al número de veces que se repiten $a_1, a_2, \ldots, a_k$, respectivamente (nota que $r_1 + r_2 + \cdots + r_k = n$). El número de permutaciones de tamaño n de A es:

$$\frac{n!}{r_1! r_2! \ldots r_k!}$$

¿Por qué funciona esta fórmula?

Existen n lugares para colocar los n elementos de A. Acomodemos primero los objetos del tipo a_1: de los n lugares de los que disponemos, contamos todos los posibles subconjuntos de tamaño r_1, pues éstas serán las formas en que podemos darle lugar a los objetos del tipo a_1. Esto es $\binom{n}{r_1}$.

Restarán $n - r_1$ lugares. De ellos, contamos todos los subconjuntos de tamaño r_2 para encontrar de cuántas formas podemos acomodar los objetos del tipo a_2, es decir, $\binom{n-r_1}{r_2}$. Ahora bien, para cada acomodo de los objetos a_1, debemos considerar todos los posibles acomodos para los del tipo a_2, por lo que, por el principio fundamental de conteo, ambos tipos de objetos pueden configurarse de $\binom{n}{r_1}\binom{n-r_1}{r_2}$.

Para colocar los r_3 objetos del tipo a_3 hay $n - (r_1 + r_2)$ espacios disponibles, así que existen $\binom{n-(r_1+r_2)}{r_3}$ formas de hacerlo, y $\binom{n}{r_1}\binom{n-r_1}{r_2}\binom{n-r_1-r_2}{r_3}$ configuraciones para los primeros tres tipos de objetos.

Repitiendo sucesivamente este proceso, la siguiente expresión representa las formas de acomodar los n objetos:

$$\binom{n}{r_1}\binom{n-r_1}{r_2}\binom{n-(r_1+r_2)}{r_3}\cdots\binom{n-(r_1+r_2+\cdots+r_{k-1})}{r_k}$$

$$= \frac{n!}{r_1!(n-r_1)!} \cdot \frac{(n-r_1)!}{r_2!(n-(r_1+r_2))!} \cdots \frac{(n-(r_1+r_2+\cdots+r_{k-1}))!}{r_k!(n-(r_1+r_2+\cdots+r_{k-1}+r_k))!}$$

$$= \frac{n!}{r_1!\cancel{(n-r_1)!}} \cdot \frac{\cancel{(n-r_1)!}}{r_2!\cancel{(n-(r_1+r_2))!}} \cdots \frac{\cancel{(n-(r_1+r_2+\cdots+r_{k-1}))!}}{r_k!(n-(r_1+r_2+\cdots+r_{k-1}+r_k))!}$$

$$= \frac{n!}{r_1! \cdot r_2! \cdot r_3! \cdots r_k! \cdot (n-(r_1+r_2+\cdots+r_k))!}$$

$$= \frac{n!}{r_1! \cdot r_2! \cdots r_k! \cdot 0!} = \frac{n!}{r_1! \cdot r_2! \cdots r_k!}$$

Al número $\frac{n}{r_1! \cdot r_2! \cdots r_k!}$ se le conoce como *coeficiente multinomial* y también se representa como:

$$\binom{n}{r_1, r_2, \ldots, r_k}$$

Ejemplo 19. ¿Cuántas cadenas distintas se pueden formar con todas las letras de la palabra PROGRAMADORA?

Solución. Primero, contemos cuántas letras repetidas existen entre las 12 letras de PROGRAMADORA:

P: 1
R: 3
O: 2
G: 1
A: 3
M: 1
D: 1

Para dar lugar a la P hay 12 posibles posiciones, así que debemos contar todos los subconjuntos de tamaño 1 de un conjunto de tamaño 12, es decir, $\binom{12}{1}$; luego buscamos los posibles lugares para las R's. Dado que ya ha sido ocupado una por la P, ahora sólo tenemos 11 opciones, y como tenemos 3 R's que acomodar, podemos hacer esto de $\binom{11}{3}$ maneras. Para ubicar las O's, restan $12-(1+3)$ lugares, o sea que se puede hacer de $\binom{8}{2}$ formas (pues disponemos de dos O's). Luego, para acomodar la G habrá

$\binom{6}{1}$ opciones; para la A $\binom{5}{3}$; para la M $\binom{2}{1}$; y para la D habrá únicamente $\binom{1}{1}$ posibilidad.

Entonces el número total de palabras diferentes que podemos formar es:

$$\binom{12}{1}\binom{11}{3}\binom{8}{2}\binom{6}{1}\binom{5}{3}\binom{2}{1}\binom{1}{1}$$

$$=\frac{12!}{1!\cdot 11!}\cdot\frac{11!}{3!\cdot 8!}\cdot\frac{8!}{2!\cdot 6!}\cdot\frac{6!}{1!\cdot 5!}\cdot\frac{5!}{3!\cdot 2!}\cdot\frac{2!}{1!\cdot 1!}\cdot\frac{1!}{1!\cdot 0!}$$

$$=\frac{12!}{1!\cdot \cancel{11!}}\cdot\frac{\cancel{11!}}{3!\cdot \cancel{8!}}\cdot\frac{\cancel{8!}}{2!\cdot \cancel{6!}}\cdot\frac{\cancel{6!}}{1!\cdot \cancel{5!}}\cdot\frac{\cancel{5!}}{3!\cdot \cancel{2!}}\cdot\frac{\cancel{2!}}{1!\cdot \cancel{1!}}\cdot\frac{\cancel{1!}}{1!\cdot 0!}$$

$$=\frac{12!}{1!\cdot 3!\cdot 2!\cdot 1!\cdot 3!\cdot 1!\cdot 1!}=\binom{12}{1,3,2,1,3,1,1}$$

Como puedes imaginar, es mucho más eficiente y rápido programar una división de factoriales que el producto de varios coeficientes binomiales.

Ejemplo 20. Dada una cadena de letras mayúsculas conformada únicamente por caracteres del alfabeto latino, obtén la cantidad de todas las posibles permutaciones que se pueden obtener de dicha cadena.

Solución. Después de recibir la cadena, hay que identificar todas las letras distintas que la componen, y contabilizar cuántos de esos caracteres se tienen en ella, para luego aplicar la fórmula de coeficientes multinomiales que obtuvimos previamente.

```
1  //Creamos la función que devuelva el Multinomial.
2  int multinomial(int n, vector<int> V)
3  {
4      int res = factorial(n);
5      for (int k:V)
6          res = res / factorial(k);
7      return res;
8  }
9
```

```
10  int solucion()
11  {
12      string cadena;
13      vector <int> V(26); //Para las letras (se inicializa en 0).
14      cin>>cadena;
15
16      //Contamos las veces que cada letra se repite.
17      for (char letra:cadena)
18          V[letra - 'a'] = V[letra - 'a'] + 1;
19
20      //Llamamos a la función "multinomial" que creamos arriba.
21      return multinomial(cadena.size(), V);
22  }
```

En este ejemplo introducimos el fragmento de programa correspondiente a la obtención de un coeficiente multinomial.

Ejemplo 21. Dados los enteros positivos n, d_1 y d_2, donde $d_1 + d_2 \leq n$, obtén la cantidad de números de n dígitos que se pueden obtener con exactamente d_1 1's y d_2 9's en los que, además, no aparezca ningún cero.

Solución. Primero elegimos los d_1 lugares para los 1's: $\binom{n}{d_1}$. De los lugares restantes, elegimos los d_2 lugares para los 9's: $\binom{n-d_1}{d_2}$. Finalmente, para el resto de los lugares $(n-d_1-d_2)$ podemos colocar cualquier dígito entre 3 y 9, inclusive, es decir, se tienen 7 opciones para cada uno. Así, obtenemos la expresión

$$\binom{n}{d_1}\binom{n-d_1}{d_2}7^{n-d_1-d_2}$$

```
1  int solucion()
2  {
3      int n, d1, d2;
4      cin>>n>>d1>>d2;
5      return binomial(n, d1) * binomial(n-d1, d2) * potencia(7, n-d1-d2);
6  }
```

Ejemplo 22. Retomando el ejemplo 16, éste también puede ser resuelto como un problema de permutaciones con repeticiones.

Solución. Analicemos: Tenemos $m+n$ objetos, de los cuales m son iguales entre sí, y n son iguales entre sí. Como habíamos dicho ya, nos interesa saber de cuántas formas distintas se pueden permutar estos objetos (considerando las repeticiones que se presentan). Así, utilizando un coeficiente multinomial:

$$\frac{(m+n)!}{m! \cdot n!} = \binom{m+n}{m, n}$$

```
1  int solucion()
2  {
3      int m, n;
4      cin>>m>>n;
5      return multinomial(m + n, {m,n});
6  }
```

Probemos que la expresión que encontramos aquí es equivalente a la que obtuvimos en la solución del ejemplo 16.

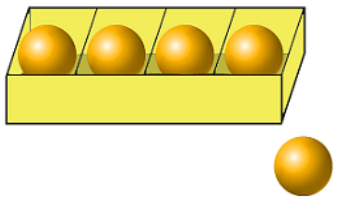
$$\frac{(m+n)!}{m! \cdot n!} = \frac{(m+n)!}{m! \cdot [(m+n)-m]!} = \binom{m+n}{m}$$

1.4. Principio de Casillas

Lo que estudiaremos en este apartado se basa en un hecho tan sencillo como lo es el afirmar que 5 es mayor que 4, no obstante, tiene importantes implicaciones y veremos ejemplos que no resultan ser tan triviales como lo parecerá el Principio de Casillas.

Veremos primero una versión "sencilla" del Principio del Palomar (como también es conocido), y posteriormente nos introduciremos a una versión generalizada, que nos brindará la oportunidad de resolver problemas un tanto más complejos.

Si tenemos m objetos para acomodar en n casillas y $m > n$, en al menos una casilla habrá más de un objeto.



Es obvio que si en cada casilla metemos exactamente un objeto nos enfrentaremos al problema de que habrá uno, o más, objetos que no serán asignados a ninguna casilla. Así, para otorgarles un lugar, es necesario colocarlos en alguna casilla que ya tiene un objeto. (Claramente no podemos empezar por colocar 0 objetos en alguna casilla, pues desde el principio nos sobrarían más objetos.)

Aunque el principio de casillas no brinda información acerca de cuál (o cuáles) es la casilla que contiene más de un objeto, este concepto nos lleva a conclusiones interesantes, como lo veremos en los ejemplos a continuación.

Ejemplo 23. En una caja hay 20 ratones: 10 hembras y 10 machos. ¿Cuántos ratones es necesario sacar para asegurar que afuera hay al menos un macho y una hembra?

> **Solución.** El peor de los casos se presenta cuando, uno tras otro, sacamos a los 10 ratones del mismo sexo. Así, para asegurar que haya al menos una hembra y un macho es necesario sacar 11 ratones, pues aun en el peor de los casos pasaría que el onceavo ratón es del sexo contrario.

Ejemplo 24. La entrada de este problema será un número natural n seguido de una serie de n enteros positivos $a_1, a_2, \ldots, a_n$ no necesariamente consecutivos.

El reto será encontrar una subserie de números de la forma $a_i, a_{i+1}, a_{i+2}, \ldots, a_{j-1}, a_j$ cuya suma sea divisible por n, e imprimir el valor de i y de j. En caso de que no exista un subconjunto con estas características, imprime "Inexistente".

Por ejemplo, sea $n = 5$ y la serie de números 9, 4, 3, 1, 6. La suma de la subserie 3, 1, 6 es 10, y como $10 = 5 \times 2$, existe al menos una subserie que cumple con lo pedido, y para este caso $i = 3$ y $j = 5$, pues 3 está en la tercera posición y 6 en

la quinta; aunque $9 + 1 = 10$, ésta no es una serie válida pues no son números con posiciones contiguas. (Problema modificado, tomado de [6]).

Solución. Probar todas las posibles sumas de todos los posibles tamaños no será una solución eficiente, pues tendríamos que hacer 2^n sumas (más adelante veremos por de dónde sale este número), así que busquemos otra alternativa para resolver este problema.

Pensemos primero en las siguientes n sumas:

$$\begin{aligned} S_1 &= a_1 \\ S_2 &= a_1 + a_2 \\ S_3 &= a_1 + a_3 + a_3 \\ &\vdots \\ S_n &= a_1 + a_2 + \cdots + a_n \end{aligned}$$

Si alguna de las sumas anteriores es múltiplo de n, habremos encontrado el conjunto buscado, pero debemos considerar el caso en que no sea así.

Existen n residuos posibles al dividir cualquier entero entre n: 0, 1, 2, $\ldots$, $n - 1$. Supongamos que ninguna de las sumas anteriores es múltiplo de n, y que, por lo tanto, ninguna de ellas devuelve residuo 0. Así, todas esas sumas tendrán un residuo entre 1 y $n - 1$. (Los residuos como la representación de otros enteros es trabajado en la sección de *Aritmética Modular* del capítulo de *Teoría de Números* de esta obra).

Podemos ver a los residuos posibles como las casillas del problema, y a $S_1, S_2, \ldots S_n$ como los objetos a acomodar. Entonces, dado que son n sumas y únicamente $n - 1$ residuos distintos, por el Principio de Casillas, *existen al menos dos sumas con el mismo residuo*. Digamos que dichas sumas son S_i y S_j, para $i < j$.

Ya que S_i y S_j arrojan el mismo residuo al ser divididas entre n, podemos expresar a S_1 y a S_2 como $(k_1 \cdot n + r)$ y como $(k_2 \cdot n + r)$, respectivamente.

Restándolos, obtenemos:

$$S_j - S_i = (k_2 \cdot n + r) - (k_1 \cdot n + r) = n(k_2 - k_1)$$

De aquí, sabemos que n divide a la diferencia de S_i y S_j, lo que significa que n divide a $(a_1 + a_2 + \cdots + a_i + a_{i+1} + \cdots + a_j) - (a_1 + a_2 + \cdots + a_i) = a_{i+1} + a_{i+2} + \cdots + a_j$. Y así aseguramos que, o n divide a alguna S_k que consideramos al inicio de la solución, o bien, que n divide a la diferencia de dos sumas con el mismo residuo módulo n, así que en ningún caso se tendrá que imprimir "Inexistente".

```
pair<int, int> solucion()
{
    int i, n;
    cin>>n;
    vector <int> a(n + 1), suma(n + 1);
    for (i = 1; i <= n; i = i + 1)
        cin>>a[i];

    //Obtenemos el residuo que arrojan las S_i al dividirlas por n.
    for (i = 1; i <= n; i = i + 1)
        suma[i] = (suma[i-1] + a[i]) % n;

    //Si alguna S_i es múltiplo de n, la salida será 1 e i.
    for (i = 1; i <= n; i = i + 1)
        if (suma[i] == 0)
            return {1, i};

    map<int, int> residuo_repetido;

    //Si el residuo de suma[i] ya pasó, la respuesta es él e i.
    //Si no ha pasado, se almacena en el mapa y seguimos buscando.
    for (i = 1; i <= n; i = i + 1)
        if (residuo_repetido[ suma[i] ])
            return {residuo_repetido[ suma[i] ], i};
        else
            residuo_repetido[ suma[i] ] = i;
}
```

Ejemplo 25. Dados los enteros positivos a y b, donde $a < b$, encuentra el momento en que la expansión decimal de $\frac{a}{b}$ se vuelve periódica, e imprime el tamaño de su periodo. (Problema modificado, tomado de [7]).

Solución. Ya que sabemos que a y b son enteros, podemos asegurar que el número $\frac{a}{b}$ es un número racional, y por definición, todo número racional, o tiene un periodo o es finito. Para este último caso, decimos que su periodo es $\bar{0}$, y que, por tanto, el tamaño de su periodo es 1. De esta forma, podemos considerar a todo número racional como infinito.

Dado que el número $\frac{a}{b}$ es infinito, al realizar el conocido algoritmo de la división (mismo que seguramente aprendiste en la primaria), éste podrá ser hecho de manera infinita, pues siempre habrá un residuo (aunque sea 0, cuenta).

Únicamente podemos obtener residuos entre 0 y $b-1$, inclusive, y podemos verlos como las casillas de nuestro problema. Por otro lado, es posible considerar a los residuos que se van obteniendo de la división usual como los objetos a acomodar.

Así, por el principio de casillas, ya que tenemos más objetos que casillas, es seguro que se repetirá al menos uno de los residuos. Dado que el número que se le agrega al residuo es siempre 0 (porque $a < b$), cuando se encuentre el primer residuo repetido el periodo volverá a comenzar.

Por ejemplo, si los números dados son 57 y 110, la división lucirá así:

$$
\begin{array}{r|l}
 & 0{,}51818 \\
\hline
110 & 57 \\
 & 570 \\
 & \quad 200 \\
 & \quad\quad 900 \\
 & \quad\quad\quad 200 \\
 & \quad\quad\quad\quad 900
\end{array}
$$

57 se puede expresar como $110(0) + \mathbf{57}$.
Multiplicamos 57×10 (pues el algoritmo dicta que se debe agregar un 0), y podemos expresar a 570 como $110(5) + \mathbf{20}$.
Multiplicamos 20×10, y $200 = 110(1) + \mathbf{90}$.
Multiplicamos 90×10, y $900 = 110(8) + \mathbf{20}$.

En cada paso, lo que nos ha interesado es guardar los residuos, pues buscamos el momento en que alguno se repita. Con la última iteración, vemos que, una vez más, encontramos a 20 como residuo, por lo tanto, ahí vuelve a comenzar el periodo. Para encontrar el tamaño del periodo, únicamente es necesario revisar cuál de los residuos es el repetido y con-

tar los elementos entre los dos residuos iguales, como lo veremos en el programa a continuación.

```
int solucion()
{
    map<int, int> residuos;
    int i = 1;
    cin>>a>>b;

    //Mientras no haya aparecido el residuo obtenido:
    while (not residuos[a])
    {
        residuos[a] = i;
        i = i + 1;

        //Agregamos un 0, equivalente a multiplicar por 10.
        a = a * 10;

        //Obtenemos el nuevo residuo.
        a = a % b;
    }

    //Tamaño del periodo.
    return i - residuos[a];
}
```

1.4.1. Principio de Casillas Generalizado

Si tenemos $nk+1$ objetos para acomodar en n casillas, en al menos una casilla habrá $k+1$ (o más) objetos.

Demostración. Sean $1, 2, \ldots, n$ las casillas de las que disponemos. Procedamos a analizar el peor caso colocando en cada casilla exactamente k elementos. Así, la suma de los objetos que colocamos en las casillas será nk. Sin embargo, hay un objeto que aún no ha sido acomodado, mismo que deberá ser introducido en alguna de las casillas que ya tiene k objetos. Por tanto, al menos una casilla tendrá $k+1$ elementos.

□

Ejemplo 26. Para este problema tendremos k colores para colorear los vértices de un polígono de n lados (y n vértices). Independientemente de cómo lo pintemos, ¿cuántos vértices tiene la figura geométrica más grande que puedes asegurar que existe tal que todos sus vértices sean del mismo color?

> **Solución.** Podemos visualizar a los n vértices del polígono dado como las casillas del problema, y a los k colores como los objetos a los que pretendemos darles lugar. Así, por el Principio de Casillas Generalizado, podemos tener certeza de que habrá una figura con al menos $\frac{n}{k}$ vértices del mismo color, y como no podemos asegurar que este número sea entero, concluimos que hay una figura con al menos $\left\lceil \frac{n}{k} \right\rceil$ vértices.

```
1  int solucion()
2  {
3      int n, k;
4      cin>>n>>k;
5      return ceil(n / k);
6  }
```

1.5. Separadores

Cierto hombre de negocios se levanta un día cansado de su demandante y agotadora vida y decide vender todos sus bienes. Ha decidido que utilizará su dinero para comenzar una nueva vida, comprar una granja con animales y vivir de la tierra. Encontró una buena oferta y ya adquirió una granja, y ahora está considerando qué animales comprar. La persona que le venderá todos los animales le ha dado la oportunidad de elegir 25 ejemplares, entre borregos, cerdos, vacas y aves. ¿De cuántas formas puede elegir a sus 25 nuevos amiguitos, tomando en cuenta que puede elegir 0 (o 25) de cualquier especie?

Encontrar todas las posibles sumas de animales que resulten en 25 es un trabajo que ciertamente provoca mucha pereza, pues seguramente no serán pocos casos. En los siguientes párrafos mostraremos una forma más elegante e interesante de respon-

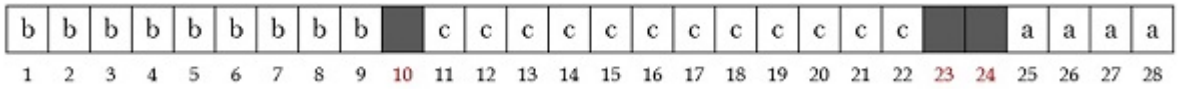
der a preguntas como éstas.

Imagina una fila de 25 cuartos: uno para cada uno de los animales que elija nuestro granjero debutante; agregaremos a esta fila 3 cuartos más, es decir, ahora tenemos 28.

Digamos que el granjero elige b borregos, c cerdos, v vacas y a aves, y pensemos que, sin importar cuántos sean, los va a meter precisamente en ese orden: en los primeros b cuartos introducirá los b borregos; la $(b+1)$-ésima habitación la dejaremos vacía. Luego será el turno de los c cerdos, y nuevamente dejaremos vacía la habitación siguiente, o sea, la $(b+c+2)$-ésima. Posteriormente damos lugar a las vacas en los siguientes v cuartos, y, una vez más, dejamos la habitación siguiente vacía (la $(b+c+v+3)$-ésima habitación). Y finalmente colocamos las a aves en los a cuartos restantes. Haciendo cuentas, podemos ver que se utilizan las 28 habitaciones.

Ahora bien, nota que cuando determinamos las posiciones de las habitaciones que permanecerán vacías, determinamos el número de ejemplares de cada especie, es decir, esas habitaciones fungen como *separadores* entre especie y especie.

Por ejemplo, para el caso en el que el granjero compra 9 borregos, 12 cerdos, 0 vacas y 4 aves, los separadores se ubicarían en habitaciones 10, 23 y 24.



Así, si determinamos todas las configuraciones que pueden tomar los separadores, estaremos contando todas las maneras en que se pueden comprar los 25 animales. La pregunta ahora es cómo determinar todos los posibles lugares para los *separadores*. Veámoslo de la siguiente manera: tenemos un conjunto de 28 habitaciones, y queremos elegir 3 de ellas. ¡Huele a coeficiente binomial! Entonces, podemos elegir las posiciones de los separadores de $\binom{28}{3}$, y éstas son las formas en que el granjero puede elegir a sus 25 animalitos.

Formalizando un poco el resultado anterior, si queremos elegir n objetos que pueden ser de k tipos distintos, podemos encontrar la cantidad de formas en que esto se puede realizar *agregando* $k-1$ objetos y eligiéndolos del conjunto de cardinalidad $n+(k-1)$, para que cumplan con una función de *separadores*. Esto nos lleva a la expresión:

$$\binom{n+(k-1)}{k-1}$$

.

Ejemplo 27. n personas pretenden entrar a una conferencia, sin embargo se ha tomado la decisión de abrir la misma conferencia televisada en varias salas, pues el número de asistentes excede la capacidad del lugar en que originalmente se había pensado dar. Si hay 6 salas disponibles a las que las personas ingresarán *siguiendo el orden de una fila*, ¿de cuántas formas es posible ingresar a los asistentes, si consideramos válido que haya salas vacías?

Solución. Imaginemos que le damos un lugar sobre una fila a cada personas. A dicha fila le agregamos cinco lugares que luego elegiremos para delimitar seis subconjuntos.

Así, podemos resumir la respuesta con la expresión

$$\binom{n+(6-1)}{6-1} = \binom{n+5}{5}$$

```
int solucion()
{
    int n;
    cin>>n;
    return binomial(n + 5, 5);
}
```

Ejemplo 28. Sean a, b, c, d enteros no negativos. ¿Cuántas tuplas existen que satisfagan la igualdad $a+b+c+d=n$ para algún $n \in \mathbb{N}$ dado?

Solución. Para este problema es conveniente ver a n no como un número, sino como su representación en n objetos. Así, el problema se traduce a elegir cuántas unidades agruparemos en a, cuántas en b, cuántas en c y

cuántas en d.

Ya que disponemos de 4 tipos de objetos (a, b, c y d), debemos agregar a n tres unidades y luego elegir los separadores. Esto es $\binom{n+3}{3}$.

```
int solucion()
{
    int n;
    cin>>n;
    return binomial(n + 3, 3);
}
```

Ejemplo 29. Sean a, b, c, d números naturales. ¿Cuántas tuplas existen que satisfagan la igualdad $a + b + c + d = n$ para un $n \in \mathbb{N}$ dado?

Solución. La diferencia entre este ejemplo y el anterior podría parecer sin importancia, sin embargo, el hecho de no incluir al 0 en los valores que pueden tomar a, b, c y d le brinda una dificultad extra al problema.

Para asegurar que la presencia del 0 no nos molestará, es necesario darle a los 4 números de la tupla que buscamos el menor valor posible, esto es 1.

Disponíamos de n unidades, sin embargo, ya que debemos asegurar que a, b, c y d sean mayores que 0, agregaremos a cada uno una unidad, con lo que únicamente nos quedaremos con $n - 4$ de los objetos que teníamos inicialmente. De esta forma, ahora debemos preguntarnos cómo podemos repartir esas $n - 4$ unidades restantes, lo que puede ser tratado como un problema tradicional de separadores, es decir, se resolverá con la expresión:

$$\binom{(n-4)+(4-1)}{4-1} = \binom{n-1}{3}$$

.

```
int solucion()
{
    int n;
    cin>>n;
    return binomial(n - 1, 3);
}
```

Proposición 1. El número de tuplas $< r_1, r_2, \ldots, r_k >$ que se pueden formar de tal forma que cada r_i sea un número natural y que $\sum_{i=1}^{k} r_i = n$, para $n \in \mathbb{N}$ dado, es:

$$\binom{n-1}{k-1}$$

Demostración. Para asegurar que cada r_i sea mayor que 0, agregamos una unidad a cada uno (como lo hicimos en el ejemplo 29), con lo que restan $n-k$ unidades, mismas que debemos repartir de alguna forma entre los k r_i's de la tupla.

Para contar de cuántas formas podemos hacer esa repartición usamos separadores: A las $n-k$ unidades que nos quedan le sumamos la cantidad de unidades que debemos agregar para trabajar con separadores, esto es $(n-k)+(k-1)$, y contamos de cuántas formas podemos elegirlos:

$$\binom{(n-k)+(k-1)}{k-1} = \binom{n-1}{k-1}$$

□

Ejemplo 30. Sea n un número natural mayor que 2. ¿Cuántos números naturales de n dígitos existen tales que su expansión decimal es $a_1 a_2 \ldots a_n$, donde o $a_i \leq a_j$ para todo $i < j$ (es decir, los dígitos son no descendentes), o bien $a_i \geq a_j$ para todo $i < j$ (no ascendentes), si todos sus dígitos deben ser mayores que 0 y menores o iguales que n?

Por ejemplo, si $n = 4$ los números 1234, 1123 y 4333 son válidos, mientras que 1432 no lo es.

Solución. Separaremos el problema en casos: Los números que únicamente contienen un dígito, los que contienen dos dígitos distintos, ..., los que contienen n dígitos distintos.

Para el caso en el que usamos un sólo dígito la respuesta es trivial, pues habrá tantos números como dígitos de los que disponemos, o sea, n; el caso en el que se usan los n dígitos también es muy sencillo, ya que sólo

habrá dos: $123\ldots n$ y $n\ldots 321$.

Concentrémonos pues, en el resto de los casos.

Digamos que los números que formemos contendrán exactamente k dígitos distintos, para $k = 2, 3, \ldots, n-1$. Lo primero que haremos será contar de cuántas formas podemos elegir estos k dígitos de entre los n que tenemos, esto es $\binom{n}{k}$.

Ahora contaremos únicamente los números de dígitos no descendentes. Es fácil ver que una vez elegidos los dígitos que participarán en el número, no importa cuántos sean de cada uno, siempre tomarán el mismo orden (de menor a mayor), así que el problema ahora se reduce a contar cuántos dígitos de cada tipo se pueden introducir al número en construcción, lo que podemos hacer con separadores: Disponemos de k dígitos distintos (de cada uno, una cantidad ilimitada) y pretendemos llenar n lugares con ellos de tal forma que todos los dígitos se presenten al menos una vez (si no fuera así, estaríamos contando números que utilizan menos de k dígitos distintos). Por la proposición 1, podemos hacer esto de $\binom{n-1}{k-1}$ formas.

Finalmente, debemos contar los números cuyos dígitos son no ascendentes, pero esto será fácil si notas que por cada número de dígitos no ascendentes existe exactamente uno de dígitos no descendentes al obtener su número "espejo".

Así, si tenemos k dígitos distintos, existen $\binom{n}{k}\binom{n-1}{k-1}(2)$ números de la forma que buscamos.

Luego, ya que k toma valores desde 2 hasta $n-1$, obtenemos la suma:

$$2\sum_{k=2}^{n-1}\binom{n}{k}\binom{n-1}{k-1}$$

A la que debemos sumar los $n+2$ números que resultan cuando $k = 1$ y cuando $k = n$. Por lo tanto, la fórmula que resuelve el problema es:

$$n + 2 + 2\sum_{k=2}^{n-1} \binom{n}{k}\binom{n-1}{k-1}$$

```
int solucion()
{
    int n, k, respuesta;
    cin>>n;

    //Cuando k = 1 y k = n.
    respuesta = n + 2;

    //El resto de los valores que tomará k.
    for (k = 2; k <= n - 1; k = k + 1)
        respuesta = respuesta + 2*binomial(n, k)*binomial(n-1, k-1);

    return respuesta;
}
```

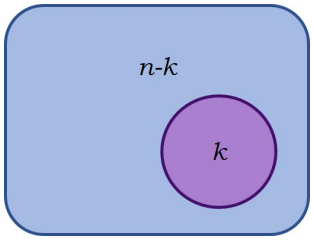
1.6. Coeficientes Binomiales

Hasta este momento hemos estudiado cómo los coeficientes binomiales son útiles para realizar conteos inteligentes. En esta sección discutiremos algunas propiedades interesantes de los mismos que mostrarán lo potentes que son estos entes matemáticos.

Proposición 2. Sean n y k enteros no negativos tales que $0 \leq k \leq n$. Se cumple entonces que:

$$\binom{n}{k} = \binom{n}{n-k}$$

Demostración. Pensemos en un conjunto A de cardinalidad n. Para contar todos los subconjuntos de A de tamaño k utilizamos la expresión $\binom{n}{k}$, sin embargo, tomar esos k objetos no es la única elección que llevamos a cabo, pues elegir los elementos que formarán parte de un subconjunto de tamaño k implica elegir aquéllos que no estarán dentro de él, es decir, elegir $n-k$ objetos de A, lo que se traduce a $\binom{n}{n-k}$. □



1.6.1. Teorema del Binomio de Newton

Teorema 1. Sean a y b dos números cualesquiera. Para todo $n \in \mathbb{N}$ se cumple:

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^{n-i} b^i$$

Demostración. La expresión $(a+b)^n$ se puede escribir como un producto de n términos iguales a $a+b$. Así, cuando aplicamos la ley distributiva a dicha multiplicación, debemos elegir, para cada uno de esos factores, si tomar a a o a b. De este modo, obtendremos una suma de 2^n términos de la forma $a^{n-k}b^k$, donde k tomará todos los valores enteros entre 0 y n. Por ejemplo, si $n = 2$, $(a+b)^2 = (a+b)(a+b) = aa+ab+ba+bb$, llegando así a una expresión de 2^2 sumandos de la forma $a^{n-k}b^k$.

Luego agrupamos los términos semejantes. Para contarlos, veamos que si en un sumando hay k b's, cada una tendrá una posición, y con ello, la posición de las $n-k$ a's se determinará también. Lo anterior implica elegir k posiciones de las n que hay en total, lo que nos lleva a ver que existirán $\binom{n}{k}$ términos semejantes.

Así, al agrupar los términos semejantes, obtendremos sumandos de la forma:

$$\binom{n}{k} a^{n-k} b^k$$

Y contemplando que k tomará los valores $0, 1, 2, \ldots, n$, llegamos a la suma:

$$(a+b)^n = \binom{n}{0}a^n + \binom{n}{1}a^{n-1}b + \binom{n}{2}a^{n-2}b^2 + \cdots + \binom{n}{n}b^n$$

$$= \sum_{i=0}^{n}\binom{n}{i}a^{n-i}b^i$$

□

La igualdad que nos brinda la proposición 2 nos ayuda a obtener una forma alternativa de representar el teorema del binomio.

$$(a+b)^n = \sum_{i=0}^{n}\binom{n}{n-i}a^{n-i}b^i$$

Corolario 1. Sea n un número natural y a un número real. Entonces:

$$(a+1)^n = \sum_{i=0}^{n}\binom{n}{i}a^i = \sum_{i=0}^{n}\binom{n}{n-i}a^i$$

Demostración. Directa del teorema 1 y del hecho de que para todo i será cierto que $1^{n-i} = 1$.

□

Ejemplo 31. Dado un número real a y un número natural n, imprime la expresión que resulta del binomio $(a+b)^n$.

Solución. Basta con aplicar en el algoritmo el teorema del binomio recién estudiado: Debemos operar con a y mantener a b como una variable.

```
1  string solucion()
2  {
3      string binomio="";
4      int n, i;
5      double a;
6      cin>>n>>a;
```

```
7
8       for (i = 0; i <= n; i = i + 1)
9       {
10          if (i == 0)
11          {
12              binomio = binomio + to_string(a) + "b^" + to_string(i);
13              continue;
14          }
15          binomio = binomio + " + " + to_string( binomial(n, i)
16                  * pow(a, n - i) ) + "b^" + to_string(i);
17      }
18
19      return binomio;
20  }
```

Ejemplo 32. Dado un número natural n y un entero k tal que $0 \leq k \leq n$, imprime el k-ésimo sumando que resulta de resolver la expresión $(a+b)^n$.

Solución. El teorema del binomio dicta que $(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^{n-i} b^i$. Como puedes ver, la suma comienza a iterar en 0, es decir, el primer sumando surge cuando i vale 0, el segundo cuando $i = 1$, el tercero cuando $i = 2$, y así sucesivamente. Así, obtenemos el k-ésimo término cuando $i = k - 1$. Por lo tanto, debemos imprimir:

$$\binom{n}{k-1} a^{n-(k-1)} b^{k-1}$$

```
1   string solucion()
2   {
3       int n, k;
4       cin>>n>>k;
5       return to_string( binomial(n, k - 1) ) + "a^"
6              + to_string(n - k + 1)+ "b^" + to_string(k - 1);
7   }
```

Es posible generalizar el teorema del Binomio a condiciones en las que existen más de dos variables en cuestión, para lo cual nos ayudarán los coeficientes multinomiales que antes vimos.

Teorema 2. Para $n \in \mathbb{N}$ y $a_i \in \mathbb{R}$,

$$(a_1 + a_2 + \cdots + a_t)^n = \sum \binom{n}{n_1, n_2, \ldots, n_t} a_1^{n_1} a_2^{n_2} \cdots a_t^{n_t}$$

donde la suma se realiza sobre todas las tuplas de enteros no negativos $< n_1, n_2, \ldots, n_t >$ cuya suma es n.

Demostración. Replicaremos la demostración usada en el teorema del binomio de Newton ajustándola a nuestras condiciones actuales.

La expresión $(a_1 + a_2 + \cdots + a_t)^n$ es el producto de n expresiones idénticas a $a_1 + a_2 + \cdots + a_t$. Aplicando la ley distributiva a dicho producto, obtendremos una suma de productos en los que para cada factor debemos elegir algún a_i. Así, obtendremos multiplicaciones de la forma $a_1^{n_1} a_2^{n_2} \cdots a_t^{n_t}$, donde $\sum_{i=1}^{t} n_i = n$.

Para agrupar los términos semejantes, contamos de cuántas formas podemos elegir a los factores a_1 de los n paréntesis, esto es, $\binom{n}{n_1}$. De los sobrantes $(n - n_1)$, elegimos los factores de la forma a_2, es decir, $\binom{n-n_1}{n_2}$. Aplicando el mismo razonamiento y el principio fundamental de conteo, se pueden elegir los t factores distintos de:

$$\binom{n}{n_1}\binom{n - n_1}{n_2} \cdots \binom{n - n_1 - \cdots - n_{t-1}}{n_t}$$

formas, pero en la sección de *Permutaciones con Repeticiones*, probamos que lo anterior es igual a

$$\frac{n!}{n_1! \cdot n_2! \cdots n_t!}$$

lo que es igual al multinomial indicado en el teorema.

Concluyendo, aprovechamos el resultado anterior para llegar al siguiente punto:

$$(a_1 + a_2 + \cdots + a_t)^n = \sum \binom{n}{n_1, n_2, \ldots, n_t} a_1^{n_1} a_2^{n_2} \cdots a_t^{n_t}$$

donde la suma se extiende a todas las tuplas $< n_1, n_2, \ldots, n_t >$ cuya suma es n.

□

Ejercicio. Dados cuatro números reales a_1, a_2, a_3 y a_4, y un número natural n, devolver $(a_1 + a_2 + a_3 + a_4)^n$.

Solución.

```
double solucion()
{
    double a1, a2, a3, a4;
    int n, n1, n2, n3, n4;
    double respuesta = 0.0;
    cin>>a1>>a2>>a3>>a4>>n;

    //Para obtener todas las tuplas <n1,n2,n3,n4>.
    for (n1 = 0; n1 <= n; n1 = n1 + 1)
        for (n2 = 0; n2 <= n - n1; n2 = n2 + 1)
            for (n3 = 0; n3 <= n - n1 - n2; n3 = n3 + 1)
            {
                n4 = n - n1 - n2 - n3;
                respuesta = respuesta + multinomial(n, {n1,n2,n3,n4})
                          * potencia(a1, n1) * potencia(a2, n2)
                          * potencia(a3, n3) * potencia(a4, n4);
            }

    return respuesta;
}
```

1.6.2. Triángulo de Pascal

Muy probablemente ya conozcas el Triángulo de Pascal y estés familiarizado con su relación con el teorema de Binomio. Éste es un acomodo triangular de números naturales en los que los elementos de las orillas laterales son 1 y el resto son la suma de los dos enteros que se encuentran inmediatamente arriba. En la siguiente ilustración puedes ver los primeros 5 renglones del Triángulo de Pascal.

$$
\begin{array}{ccccccc}
 & & & 1 & & & \\
 & & 1 & & 1 & & \\
 & 1 & & \mathbf{2} & & 1 & \\
1 & & \mathbf{3} & & 3 & & 1 \\
 & 4 & & \mathbf{6} & & 4 & \\
 & & & \cdots & & &
\end{array}
$$

Por ejemplo, 2 es la suma de $1+1$, 3 es la suma de $1+2$ y 6 es la suma de $3+3$; los enteros de ambos márgenes laterales son todos 1's.

Ahora bien, si deseamos obtener el desarrollo de $(a+b)^n$, sabemos que el comportamiento de los exponentes de a y de b es decrementar e incrementar en 1 cada vez, respectivamente, como lo estudiamos en el teorema 1. Para conocer el coeficiente numérico de cada producto $a^{n-k}b^k$, basta con revisar el $(n+1)$-ésimo renglón del triángulo de Pascal, mismo que contendrá $n+1$ constantes, que se le asignarán, en el orden en que se encuentran, a cada producto $a^{n-k}b^k$. Es decir, el triángulo de Pascal nos ayuda a saber cuántos términos semejantes hay de cada factor obtenido. Por ejemplo, para obtener $(a+b)^3$ sabemos que todos los productos distintos conformados por a y b son a^3, a^2b, ab^2 y b^3. Vamos al cuarto renglón del triángulo de Pascal para obtener sus términos numéricos y concluimos que el desarrollo de $(a+b)^3$ es $a^3+3a^2b+3ab^2+b^3$.

Ahora que sabemos que el triángulo de Pascal aporta una forma de obtener los coeficientes numéricos del desarrollo de un binomio potenciado, y que sabemos que esos mismos términos pueden ser escritos como coeficientes binomiales (como estudiamos en el teorema 1), podemos establecer una equivalencia entre ellos, lo que es más, podemos recrear al triángulo de Pascal en términos de coeficientes binomiales. Así, en el $(n+1)$-ésimo renglón del triángulo hallaremos los coeficientes que le corresponden a $(a+b)^n$, que son:

$$\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n-1}, \binom{n}{n}$$

Por tanto, la representación en coeficientes binomales del triángulo de Pascal será:

$$\binom{0}{0}$$

$$\binom{1}{0} \quad \binom{1}{1}$$

$$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$$

$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$

$$\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4}$$

$$\dots$$

La formalización de la manera en que se pueden obtener los coeficientes binomiales a través de sumas de términos que los anteceden dentro del Triángulo de Pascal nos lleva al siguiente teorema.

Teorema 3. [Fórmula de Pascal]. Sea n un número natural y sea k un entero no negativo tales que $n \geq k+1$. Se satisface entonces la siguiente igualdad.

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Demostración. Sea A un conjunto con n elementos. Existen $\binom{n}{k}$ subconjuntos de cardinalidad k de A.

Luego, sea x algún elemento en particular de A. Al formar los subconjuntos de tamaño k (cuyo total ya sabemos que es $\binom{n}{k}$), tenemos dos opciones: incluir o no a x. Si lo incluimos, restan $n-1$ objetos de los cuales hay que escoger $k-1$ elementos para completar un subconjunto de cardinalidad k, esto se puede hacer de $\binom{n-1}{k-1}$ maneras. Por el contrario, cuando x no es incluido, restan $n-1$ objetos de los que elegiremos los k que necesitamos, lo que se resume a $\binom{n-1}{k}$ subconjuntos distintos.

Finalmente, ya que crear los subconjuntos de tamaño k depende de elegir o no a x, la totalidad de dichos subconjuntos será la unión de lo que resulte en ambos escenarios: $\binom{n-1}{k-1}$ cuando no es tomado y $\binom{n-1}{k}$ en el caso opuesto, con lo que concluimos que:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

□

Con lo anterior, encontramos un comportamiento recursivo en los coeficientes binomiales. Esta propiedad nos ayudará más adelante, cuando abordemos *Programación Dinámica*.

Ejemplo 33. Retomaremos el ejemplo 16 que vimos en la sección de *Combinaciones*, y lo abordaremos desde un punto de vista recursivo.

Recordando, tenemos una cuadrícula de $m \times n$ y deseamos llegar del extremo inferior derecho (A) al extremo superior izquierdo (B), para lo que únicamente son válidos movimientos hacia arriba y hacia la derecha. Con esto, buscamos la cantidad de formas en que la tarea puede realizarse.

> **Solución.** Para llegar de A a cualquier punto de la forma $(i, 0)$ existe únicamente una forma de hacerlo: avanzar de manera horizontal. Luego, ya que i tomará todos los valores enteros entre 0 y m, podemos expresar el número de caminos de $(0, 0)$ a $(i, 0)$ como $\binom{0}{0}$, $\binom{1}{0}$, $\binom{2}{0}$, $\ldots$, $\binom{m}{0}$. Lo mismo sucede con los puntos de la forma $(0, j)$: únicamente se puede llegar de una forma a cada uno (avanzando de manera vertical) y ésta puede ser escrita como $\binom{1}{1}$, $\binom{2}{2}$, $\ldots$, $\binom{j}{j}$.
>
> Ahora bien, nota que para llegar al punto $(1, 1)$ basta con sumar los caminos que obtuvimos desde A hasta los puntos $(0, 1)$ y $(1, 0)$, pues fuera de ellos (debido a las restricciones de los tipos de movimientos que podemos hacer) no hay otro modo de llegar a $(1, 1)$. Lo mismo sucede con el punto $(2, 1)$, en el que debemos sumar los caminos que obtuvimos hasta los puntos $(2, 0)$ y $(1, 1)$. En general, para llegar al punto (i, j) debemos sumar los caminos de los dos únicos puntos inmediatamente anteriores a partir de los cuales el punto (i, j) es alcanzable, es decir, de los puntos $(i - 1, j - 1)$ y $(i - 1, j)$.
>
> Llamemos $C_{i,j}$ al número de caminos que existen desde A al punto (i, j).

Hemos encontrado entonces una relación recursiva de la forma:

$$C_{i,j} = \begin{cases} 1 & \text{si } i = 0 \\ 1 & \text{si } j = 1 \\ C_{i-1,j-1} + C_{i-1,j} & \text{en otros casos} \end{cases}$$

Además, ya que los casos base de esta relación recursiva se pueden escribir en términos de coeficientes binomiales, el resto de la relación también, con lo que llegamos a la Fórmula de Pascal. Así, las orillas inferior e izquierda de la cuadrícula se llenarán de 1's y el resto resultarán de la suma de los dos datos obtenidos de manera inmediatamente anterior, simulando con ello el Triángulo de Pascal.

Finalmente, como buscamos el número de caminos que hay de $(0, 0)$ a (m, n), sólo imprimimos el término $C_{m+n,m} = C_{m-1,n-1} + C_{m-1,n}$ en caso de que m y n sean ambos distintos de 0, pues si $m = 0$ o $n = 0$, la respuesta será 1.

```
1  int C(int m, int n)
2  {
3      if (m == 0 or n == 1)
4          return 1;
5      return C(m - 1, n - 1) + C(m - 1, n);
6  }
```

Ejemplo 34. Un grupo de n personas pretenden crear una figura humana como se muestra abajo.

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 2 & & 3 & & \\ & 4 & & 5 & & 6 & \\ 7 & & 8 & & 9 & & 10 \\ & & & \cdots & & & \end{array}$$

Para ello se forman en una fila y la i-ésima persona toma el lugar i en el triángulo, considerando que 1 corresponde a la cima. A las personas de las orillas se les dan dos

monedas, y decimos que ésta es la cantidad de dinero de la que pueden disponer. El resto de los participantes resulta más beneficiado, pues tienen derecho a disponer del equivalente a la suma de las monedas que poseen las dos personas inmediatamente arriba de ellos. ¿A cuánto dinero tiene derecho la n-ésima persona?

Solución. Las dos monedas que poseen las personas de las orillas laterales pueden escribirse como sigue:

$$2\binom{0}{0}$$

$$2\binom{1}{0} \qquad 2\binom{1}{1}$$

$$2\binom{2}{0} \qquad\qquad 2\binom{2}{2}$$

$$2\binom{3}{0} \qquad\qquad\qquad 2\binom{3}{3}$$

$$\cdots$$

Sumando $2\binom{1}{0} + 2\binom{1}{1} = 2\left[\binom{1}{0} + \binom{1}{1}\right]$. Por la fórmula de Pascal antes probada, lo anterior es igual a $2\binom{2}{1}$. De hecho, para cualquier par de coeficientes binomiales consecutivos en el mismo renglón $2\binom{k-1}{i-1}$ y $2\binom{k-1}{i}$, se satisface lo siguiente.

$$2\binom{k-1}{i-1} + 2\binom{k-1}{i} = 2\left[\binom{k-1}{i-1} + \binom{k-1}{i}\right] = 2\binom{k}{i}$$

Ahora es necesario determinar cuál es la expresión (en términos de coeficientes binomiales) que corresponde a la n-ésima posición.

Nota que en el primer renglón del triángulo hay una persona, en el segundo hay dos y así sucesivamente. Por tanto, podemos escribir el número de personas que hay hasta el k-ésimo renglón como $\frac{k(k+1)}{2}$.

Con el fin de encontrar una primera aproximación de la posición de n, estableceremos la siguiente igualdad, suponiendo con ella que n se encuentra *al final* del k-ésimo renglón (y que, por tanto, tiene derecho a dos monedas).

$$\frac{k(k+1)}{2} = n \qquad \Longrightarrow \qquad k^2 + k - 2n = 0$$

Aplicando la fórmula general para obtener el valor de k y tomando en cuenta que estamos trabajando con números no negativos, llegamos a que

$$k = \frac{-1 + \sqrt{1+8n}}{2}$$

Como dijimos, a este punto, suponemos que $k \in \mathbb{N}$, pero ¿qué ocurre en caso contrario? Si $k \notin \mathbb{N}$, podemos asegurar que la parte entera de k es el último renglón que se completó. Así, restando

$$n - \frac{\lfloor k \rfloor (\lfloor k \rfloor + 1)}{2}$$

obtenemos los participantes que fueron acomodados en el renglón $k+1$.

Llamaremos r a la diferencia obtenida y buscaremos una manera de expresar cualquier cantidad en el triángulo en términos de coeficientes binomiales, al igual que el dinero que poseen las personas de las orillas.

Ya que estamos manejando elementos del $(k+1)$-ésimo renglón, la parte de arriba del coeficiente binomial es k, y dado que la persona con la posición n es el r-ésimo participante del renglón $k+1$, el término inferior del coeficiente binomial es $r-1$ (en términos de coeficientes binomiales, los renglones comienzan en 0).

Así, llegamos a que la posición n corresponde a la expresión $2\binom{k}{r-1}$, pero hay que ser cuidadosos con el caso en el que $r = 0$ (el elemento es el último de un renglón), pues el coeficiente binomial que proponemos no existiría y obtendríamos 0, lo que es una respuesta incorrecta; para la condición anterior incluiremos un condicional, en el que, si $r = 0$, la respuesta será $2\binom{k}{k} = 2(1) = 2$ (siguiendo el esquema del triángulo de Pascal). Para simplificar el trabajo de la computadora, incluiremos también en este condicional el caso en el que $r = 1$, pues $\binom{k}{0} = 1$.

```
1 int solucion()
2 {
```

```
3       int n, k, r;
4       cin>>n;
5
6       k = floor( (-1 + sqrt(1 + 8 * n)) / 2 );
7       r = n - k * (k + 1) / 2;
8
9       if (r == 0 or r ==1)
10          return 2;
11      return 2 * binomial(k, r - 1);
12  }
```

1.6.3. Conjunto Potencia

Imagina que tienes una caja con n objetos y quieres contar todos los subconjuntos que es posible obtener de ellos. Para esto, alineas los n objetos, y para crear cada subconjunto posible, debes decidir si incluir o no al i-esimo objeto, es decir, para cada elemento del conjunto existen dos opciones: tomarlo o no. Ya que son n objetos, por el principio fundamental de conteo, existen 2^n subconjuntos.

Digamos, por ejemplo, que tenemos el conjunto $A = \{1, 2, 3\}$. Los subconjuntos que se pueden crear son $\emptyset$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$ y $\{1, 2, 3\}$. El conjunto vacío es la elección de no tomar ninguno de los elementos de A, el subconjunto $\{1, 2\}$ implica haber elegido a 1 y a 2, pero excluido a 3, mientras que el subconjunto $\{1, 2, 3\}$ nos dice que incluimos a todos los objetos que A comprende.

El conjunto de todos los subconjuntos posibles de un conjunto de n elementos es llamado *conjunto potencia*, y su cardinalidad es 2^n; ya que la cardinalidad del conjunto potencia comprende a los subconjuntos de todos los tamaños de dicho conjunto, podemos expresarla también de la siguiente forma:

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = 2^n$$

La expresión anterior puede ser probada también por medio del teorema del Binomio, en el caso particular en que $a = b = 1$.

Ejemplo 35. Tenemos un conjunto A con n elementos. ¿Cuántos subconjuntos de tamaño par y cuántos de tamaño impar existen?

Solución. Cuando abordamos el conjunto potencia explicamos que para obtener todos los posibles subconjuntos era conveniente preguntarnos, para cada objeto, si tomarlo o no. En este caso haremos algo parecido.

Contaremos primero los subconjuntos de tamaño par: Para los primeros $n-1$ elementos, tenemos las dos opciones de las que ya hablamos, sin embargo, para el n-ésimo objeto, debemos contemplar que si ya se tomó un número par de objetos en los anteriores $n-1$ elementos, el n-ésimo objeto ya no se puede tomar. Por otro lado, si en los primeros $n-1$ objetos se ha tomado un número impar de elementos, la única opción para el n-ésimo es tomarlo; dicho de otra forma, una vez hechas las elecciones en $n-1$ de los objetos, el último elemento queda determinado y sólo hay una posibilidad para él.

Aplicamos el mismo razonamiento para los subconjuntos de tamaño impar, y concluimos que existen 2^{n-1} subconjuntos de tamaño impar y 2^{n-1} subconjuntos de tamaño par.

```
pair<int, int> solucion()
{
    int n;
    cin>>n;
    return {potencia(2, n - 1), potencia(2, n - 1)};
}
```

Para formalizar lo visto en el ejemplo anterior, veremos la siguiente proposición.

Proposición 3. Para todo número natural n, se cumple lo siguiente.

$$\binom{n}{0} + \binom{n}{2} + \binom{n}{4} + \cdots = \binom{n}{1} + \binom{n}{3} + \binom{n}{5} + \cdots = 2^{n-1}$$

Demostración. Retomando el teorema del Binomio, si $a = 1$ y $b = -1$, se satisface:

$$(1-1)^n = \binom{n}{0} - \binom{n}{1} + \binom{n}{2} - \cdots + (-1)^n \binom{n}{n}$$

$$\Longrightarrow \binom{n}{0} + \binom{n}{2} + \binom{n}{4} + \cdots = \binom{n}{1} + \binom{n}{3} + \binom{n}{5} + \cdots$$

Lo anterior nos dice que existe el mismo número de subconjuntos de tamaño par que de subconjuntos de tamaño impar. Además, sabemos que la suma de ambos es 2^n, y como la mitad de 2^n es 2^{n-1}, obtenemos nuevamente la cantidad de subconjuntos de cada tipo, y coincide con la que resultó en el ejemplo 35.

Concluyendo,

$$\binom{n}{0} + \binom{n}{2} + \binom{n}{4} + \cdots = \binom{n}{1} + \binom{n}{3} + \binom{n}{5} + \cdots = 2^{n-1}$$

□

1.6.4. Unimodalidad de los Coeficientes Binomiales

En la sección de *Triángulo de Pascal*, pudiste notar el comportamiento de los coeficientes binomiales dentro del mencionado triángulo: Mientras más a la orilla estén, más pequeños son con respecto a aquéllos que se encuentran al centro de cada renglón. Esto se debe a una propiedad llamada *unimodalidad*.

Decimos que una secuencia de números $a_0, a_1, a_2, \ldots, a_n$ es unimodal si existe (al menos) algún entero t tal que $0 \leq t \leq n$ que cumpla que $a_0 \leq a_1 \leq a_2 \leq \cdots \leq a_t$ y que $a_t \geq a_{t+1} \geq a_{t+2} \geq \cdots \geq a_n$

Ahora que entendimos qué significa que una secuencia sea unimodal, demostrémoslo para una serie de coeficientes binomiales.

Teorema 4. Sea n un entero no negativo. Si n es par, se satisface:

$$\binom{n}{0} < \binom{n}{1} < \binom{n}{2} < \cdots < \binom{n}{n/2}$$

y

$$\binom{n}{n/2} > \binom{n}{(n/2)+1} > \binom{n}{(n/2)+2} > \cdots > \binom{n}{n}$$

Mientras que si n es impar, se cumple:

$$\binom{n}{0} < \binom{n}{1} < \binom{n}{2} < \cdots < \binom{n}{(n-1)/2} = \binom{n}{(n+1)/2}$$

y

$$\binom{n}{(n+1)/2} > \binom{n}{((n+1)/2)+1} > \binom{n}{((n+1)/2)+2} > \cdots > \binom{n}{n}$$

Demostración. Tomemos dos coeficientes consecutivos $\binom{n}{k}$ y $\binom{n}{k+1}$. Si los restamos, obtenemos

$$\binom{n}{k} - \binom{n}{k+1} = \frac{n!}{k!(n-k)!} - \frac{n!}{(k+1)!(n-k-1)!}$$

$$= \frac{n!}{k!(n-k-1)!}\left[\frac{1}{n-k} - \frac{1}{k+1}\right]$$

A partir de lo anterior, existen tres opciones:

$$\binom{n}{k} < \binom{n}{k+1} \quad \Longrightarrow \quad \frac{1}{n-k} < \frac{1}{k+1} \quad \Longrightarrow \quad k < \frac{n-1}{2}$$

$$\binom{n}{k} > \binom{n}{k+1} \quad \Longrightarrow \quad \frac{1}{n-k} > \frac{1}{k+1} \quad \Longrightarrow \quad k > \frac{n-1}{2}$$

$$\binom{n}{k} = \binom{n}{k+1} \quad \Longrightarrow \quad \frac{1}{n-k} = \frac{1}{k+1} \quad \Longrightarrow \quad k = \frac{n-1}{2}$$

Si n es un número natural par, entonces podemos expresarlo como $2m$, para $m \in \mathbb{N}$, con lo que $m = n/2$. Analicemos primero qué sucede en este caso.

Si $k < \frac{n-1}{2}$, entonces $k < m - \frac{1}{2}$. Como $m - \frac{1}{2} < m$, por transitividad, $k < m$, es decir, $k < \frac{n}{2}$; lo que implica que cuando se cumple que $\binom{n}{k} < \binom{n}{k+1}$, se cumple también que $k < \frac{n}{2}$. Probamos entonces que:

$$\binom{n}{0} < \binom{n}{1} < \cdots < \binom{n}{n/2-1} < \binom{n}{n/2}$$

Si $k > \frac{n-1}{2}$, $k > m - \frac{1}{2}$. Podemos estar seguros de que k no es menor que m, así que $k \geq m$, es decir, $k \geq \frac{n}{2}$. Entonces, cuando $\binom{n}{k} > \binom{n}{k+1}$, $k \geq \frac{n}{2}$; o sea que:

$$\binom{n}{n/2} > \binom{n}{n/2+1} > \cdots > \binom{n}{n-1} > \binom{n}{n}$$

Como $k = \frac{n-1}{2}$, se cumple que $2m - 1 = 2k$, pero no existe ningún valor natural para k que cumpla dicha igualdad, por lo que no existen dos coeficientes binomiales consecutivos que sean iguales.

Por otro lado, cuando n es impar, lo podemos escribir como $2m+1$, $m \in \mathbb{N}$, y tenemos los siguientes escenarios.

Si $k < \frac{n-1}{2}$, $\binom{n}{k} < \binom{n}{k+1}$. Así,

$$\binom{n}{0} < \binom{n}{1} < \cdots < \binom{n}{(n-1)/2}$$

Si $k > \frac{n-1}{2}$, podemos implicar que $k \geq \frac{n+1}{2}$. Y llegamos a que si $\binom{n}{k} > \binom{n}{k+1}$, $k \geq \frac{n+1}{2}$. Se sigue:

$$\binom{n}{(n+1)/2} > \binom{n}{(n+1)/2+1} > \cdots > \binom{n}{n}$$

Si $k = \frac{n-1}{2}$, $k+1 = \frac{n+1}{2}$, con lo que concluimos que el único par de coeficientes binomiales consecutivos que son equivalentes es:

$$\binom{n}{(n-1)/2} \quad \text{y} \quad \binom{n}{(n+1)/2}$$

□

Corolario 2. Si n es un número natural, los coeficientes más grandes de la secuencia $\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \ldots, \binom{n}{n}$ son:

$$\binom{n}{\lfloor n/2 \rfloor} = \binom{n}{\lceil n/2 \rceil}$$

Demostración. Directa del teorema anterior.

□

Ejemplo 36. Vas a un parque de diversiones cuyos juegos están todos ordenados de forma triangular, tal como en el triángulo de Pascal, y tal como en el triángulo de Pascal, a cada juego se le asigna un coeficiente binomial indicando lo divertido que es.

La entrada serán dos números naturales n y k, donde $1 \leq k \leq n$. Si n es el número de renglones en el parque de diversiones, ¿cuál es el entero al que equivale el coeficiente binomial de la atracción más divertida del renglón k? Si en un renglón aparece más de una vez el coeficiente más grande, súmalos tantas veces como aparezcan.

Solución. Por la particular forma del triángulo de Pascal, sabemos que en el renglón k existen k coeficientes de la forma $\binom{k-1}{i}$, para $0 \leq i \leq k-1$.

Del corolario 2, sabemos que si $k-1$ es par, el coeficiente máximo es $\binom{k-1}{(k-1)/2}$, más aún, que es único (puesto que si $k-1$ es par $\lfloor (k-1)/2 \rfloor = \lceil (k-1)/2 \rceil$, a diferencia de cuando $k-1$ es impar, donde existen exactamente dos máximos: $\binom{k-1}{(k-2)/2}$ y $\binom{k-1}{k/2}$.

Por la proposición 2 y por el hecho de que $k-1-\frac{k-2}{2} = \frac{k}{2}$, podemos estar seguros de que $\binom{k-1}{(k-2)/2} = \binom{k-1}{k/2}$, por lo que si $k-1$ es impar, deberemos imprimir $2\binom{k-1}{k/2}$; en caso contrario, imprimiremos $\binom{k-1}{(k-1)/2}$.

```
1  int solucion()
2  {
3      int k;
4      cin>>k;
5      if ( (k-1) % 2 == 1 )
6          return 2 * binomial(k - 1, k / 2);
7      return binomial(k - 1, (k - 1) / 2);
8  }
```

Proposición 4. Para todo $n \in \mathbb{N}$, se cumple:

$$\sum_{i=0}^{n} i\binom{n}{i} = n \cdot 2^{n-1}$$

Demostración. Dividiremos la demostración en dos casos: cuando n es par y cuando n es impar.

Si n es impar, la suma tendrá un número par de sumandos $(n+1)$. De aquí, podemos "emparejar" a cada elemento de la primera mitad con uno de la segunda mitad, tomando parejas de la forma $k\binom{n}{k}$ y $(n-k)\binom{n}{n-k}$, para $0 \leq k \leq (n-1)/2$. Por la proposición 2, sabemos que los coeficientes binomiales de ambas expresiones son equivalentes, por lo que sumarlas resultaría en lo siguiente:

$$k\binom{n}{k} + (n-k)\binom{n}{n-k} = k\binom{n}{k} + (n-k)\binom{n}{k}$$

$$= (k+n-k)\binom{n}{k} = n\binom{n}{k}$$

De lo anterior, deducimos que podemos reescribir la igualdad establecida en la proposición como sigue:

$$\sum_{i=0}^{n} i\binom{n}{i} = \sum_{i=0}^{\lfloor n/2 \rfloor} n\binom{n}{i} = n\sum_{i=0}^{\lfloor n/2 \rfloor}\binom{n}{i} \tag{1.1}$$

La suma $\sum_{i=0}^{\lfloor n/2 \rfloor}\binom{n}{i}$ toma únicamente la mitad de los subconjuntos del conjunto potencia: los de los tamaños 1, 2, ..., $\lfloor n/2 \rfloor$, y dado que $\binom{n}{k} = \binom{n}{n-k}$, la suma de éstos es la misma que $\binom{n}{\lfloor n/2 \rfloor + 1} + \binom{n}{\lfloor n/2 \rfloor + 2} + \cdots + \binom{n}{n}$, por lo tanto,

$$\sum_{i=0}^{\lfloor n/2 \rfloor}\binom{n}{i} = \frac{2^n}{2} = 2^{n-1}$$

Sustituyendo en la expresión 1.1,

$$\sum_{i=0}^{n} i\binom{n}{i} = n \cdot 2^{n-1}$$

Veamos ahora el segundo caso: Si n es un natural par, podemos escribir la suma de la proposición como:

$$\sum_{i=0}^{n} i\binom{n}{i} = n\sum_{i=0}^{(n-2)/2}\binom{n}{i} + \frac{n}{2}\binom{n}{n/2} = n\left[\sum_{i=0}^{(n-2)/2}\binom{n}{i} + \frac{1}{2}\binom{n}{n/2}\right]$$

En la primera parte de la demostración recordamos la propiedad 2, misma que utilizamos aquí también. No obstante aquí existe un sumando sin "pareja": $\frac{1}{2}\binom{n}{n/2}$. La clave está en fijarnos que éste comprende la mitad de los subconjuntos de tamaño $n/2$, con lo que podemos asegurar que la suma dentro de los corchetes representa a

la mitad del conjunto potencia, por lo que en este caso también se satisface que la proposición.

□

Ejemplo 37. En un mundo imaginario, el rey de las matemáticas quiere dar una fiesta para n invitados, y quiere agasajarlos con regalos del tipo t_1, t_2,..., t_n. Inicialmente pide $\binom{n}{i}$ regalos del tipo t_i, pero de repente se le ocurre que, para ser todavía un mejor anfitrión, preparará $2i\binom{n}{i}$ regalos de cada tipo y le dará la misma cantidad de presentes a cada uno de sus n invitados. ¿Cuántos regalos le tocan a cada asistente a la fiesta?

Solución. La cantidad total de regalos será:

$$2\binom{n}{1} + 4\binom{n}{2} + \cdots + 2n\binom{n}{n}$$

$$=2\sum_{i=1}^{n} i\binom{n}{i} = 2\left[n \cdot 2^{n-1} - 0\binom{n}{0}\right]$$

$$=2 \cdot n \cdot 2^{n-1} = n \cdot 2^n$$

Luego, queremos que sean repartidos equivativamente entre los n invitados, esto es:

$$\frac{n2^n}{n} = 2^n$$

```
1  int solucion()
2  {
3      int n;
4      cin>>n;
5      return potencia (2, n);
6  }
```

1.6.5. Convolución de Vandermonde

Teorema 5. Sean k, n_1 y n_2 tres números naturales tales que $k \leq n_1 + n_2$. Se cumple entonces que:

$$\sum_{i=0}^{k} \binom{n_1}{i}\binom{n_2}{k-i} = \binom{n_1+n_2}{k}$$

Demostración. Sea A un conjunto de cardinalidad $n_1 + n_2$. Particionamos a A en dos subconjuntos A_1 y A_2, el primero de cardinalidad n_1 y el segundo de cardinalidad n_2 (cardinalidades que bien pueden ser 0).

Sabemos ya que si queremos elegir un subconjunto de tamaño k de A, tenemos $\binom{n_1+n_2}{k}$ formas de hacerlo.

Por otro lado, cuando particionamos al conjunto A, particionamos también a los subconjuntos de él. Así, cada posible subconjunto de tamaño k toma i elementos de la parte A_1 y $k-i$ de la parte A_2. Esto implica contar todas las posibles maneras de extraer i objetos de A_1, y para cada una, contemplar todas las formas en que se pueden tomar los $k-i$ faltantes de A_2. Por el principio fundamental del conteo, esto se hace de $\binom{n_1}{i}\binom{n_2}{k-i}$ maneras.

Luego, hay que considerar todas las posibilidades para i, lo que implicará tomar desde ninguno hasta los k objetos de A_1, llegando a la suma:

$$\sum_{i=0}^{k} \binom{n_1}{i}\binom{n_2}{k-i}$$

Así, contamos los subconjuntos de tamaño k que se pueden formar con los elementos del conjunto A de una forma distinta y concluimos que:

$$\sum_{i=0}^{k} \binom{n_1}{i}\binom{n_2}{k-i} = \binom{n_1+n_2}{k}$$

□

La demostración anterior incluye una partición del conjunto inicial en dos subconjuntos, obviamente disjuntos, no obstante, dicha partición se puede realizar en tantos subconjuntos como gustes, con lo que, en lugar de tener un producto de dos coeficientes binomiales, tendremos un producto de tantos coeficientes como particiones resulten, dando paso así a la presencia de un *coeficiente multinomial*.

Corolario 3.

$$\sum_{i=0}^{n} \binom{n}{i}^2 = \binom{2n}{n}$$

Demostración. Éste es un caso particular del teorema anterior, en el que $n_1 = n_2 = n$. Así, llegaremos a la igualdad:

$$\sum_{i=0}^{n} \binom{n}{i}\binom{n}{n-i} = \binom{2n}{n}$$

Como ya sabemos, $\binom{n}{i} = \binom{n}{n-i}$, con lo que:

$$\sum_{i=0}^{n} \binom{n}{i}^2 = \binom{2n}{n}$$

□

1.7. Inclusión y Exclusión

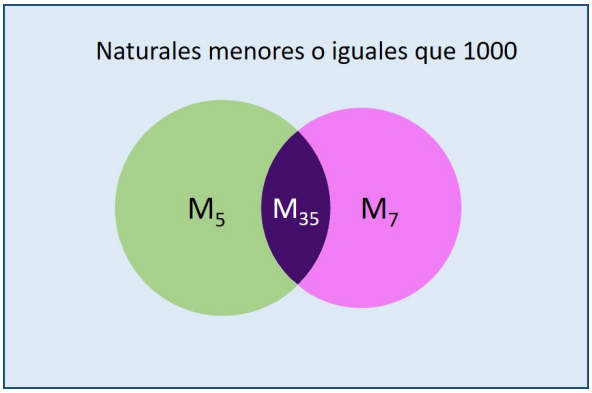
Dada una lista de los primeros 1000 números naturales, debemos tachar todos aquellos números que sean múltiplos de 5, de 7 y de ambos. ¿Cuántos números quedarán en dicha lista sin tachar?

Llamemos M_5 al conjunto de los múltiplos de 5 menores o iguales que 1000 y M_7 al conjunto de los múltiplos de 7 bajo las mismas condiciones. Ya que cada 5 naturales

aparece un múltiplo de 5, y que cada 7 aparece un múltiplo de 7, deducimos que $|M_5|$[1] $= \frac{1000}{5} = 200$ y que $|M_7| = \lfloor \frac{1000}{7} \rfloor = 142$.

Si sustraemos $|M_5|$ y $|M_7|$ de los 1000 números que tenemos inicialmente, estaremos restando los múltimos de 5 y los de 7, sin embargo, M_5 y M_7 no son conjuntos disjuntos, pues existen números que son múltiplos de ambos simultáneamente, así que dichos números están siendo quitados dos veces. Para remediar esta situación basta con sumar la cantidad de números que estén en la intersección de M_5 y de M_7.

Obtenemos la intersección de ambos conjuntos al notar que los números que son múltiplos de ambos primos tienen la forma $7(5)k$, es decir $35k$, para algún natural k. Dicho de otra forma, un número es múltiplo de 5 y de 7 si y sólo si es múltiplo de 35. Llamemos pues M_{35} a la intersección de M_5 y M_7, cuya cardinalidad será $\lfloor \frac{1000}{35} \rfloor = 28$.



Así, llegamos a la expresión $1000 - (200 + 142) + 28 = 686$. Es decir, existen 686 números que no fueron tachados. Visto en el diagrama de Venn que se muestra arriba, los representa el área azul del rectángulo.

El ejemplo que acabamos de abordar es uno de los más sencillos dentro del área de Inclusión y Exclusión, en el que la idea es partir de la cardinalidad de un conjunto y retirar los subonjuntos de los elementos que cumplen con cierta propiedad (en el caso anterior, que los números fueran múltiplos de 5 o de 7), tomando en cuenta que dichos subonjuntos no necesariamente son disjuntos, para lo que se requerirá sumar

[1] En matemáticas discretas usamos la notación |A| para hacer referencia a la cardinalidad del conjunto A, es decir, al número de elementos en él.

los objetos que hayan sido retirados más de una vez. Este procedimiento se repite de manera sucesiva tantas veces sea necesario, como se verá en la teoría en lo sucesivo.

El siguiente principio nos ayudará a generalizar este razonamiento para m subconjuntos que pueden, o no, tener elementos en común.

Teorema 6. [Principio de Inclusión y Exclusión]. Sea A un conjunto, y sean $P_1, P_2, \ldots, P_m$ propiedades que pueden tener, o no, los elementos de A. Llamamos A_i al subconjunto de los objetos que cumplan con la propiedad P_i. El número de objetos en A que no tienen ninguna propiedad es:

$$|\overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_m}| = |A| - S_1 + S_2 - S_3 + \cdots + (-1)^m S_m$$

Donde:

$$\begin{aligned}
S_1 &= |A_1| + |A_2| + \cdots + |A_m| \\
S_2 &= |A_1 \cap A_2| + |A_1 \cap A_3| + \cdots + |A_{m-1} \cap A_m| \\
S_3 &= |A_1 \cap A_2 \cap A_3| + |A_1 \cap A_2 \cap A_4| + \cdots + |A_{m-2} \cap A_{m-1} \cap A_m| \\
&\vdots \\
S_m &= |A_1 \cap A_2 \cap \cdots \cap A_m|
\end{aligned}$$

Dicho de otra forma:

$$\left|\bigcap_{i=1}^{m} \overline{A_i}\right| = |A| - \sum_{\{i\} \subset [m]} |A_i| + \sum_{\{i,j\} \subset [m]} |A_i \cap A_j| - \cdots + (-1)^m |A_1 \cap A_2 \cap \cdots \cap A_m|$$

donde la expresión $[m]$ hace referencia al conjunto $\{1, 2, 3, \ldots, m\}$.

Demostración. Del lado izquierdo de la ecuación tenemos la intersección de los complementos de todos los subconjuntos A_i, es decir, el número de elementos que no cumplen con ninguna de las propiedades P_i.

Entonces, para demostrar este teorema vamos a analizar el efecto que tiene sobre el lado izquierdo de la ecuación un elemento que cumple con n de las m propiedades, y un elemento que no cumple con ninguna.

Sea un objeto a que no cumple con ninguna de las propiedades P_i. Sabemos que éste aparece únicamente una vez en A, mientras que, por no cumplir ninguna propiedad, en S_1 aparece 0 veces, al igual que en el resto de las sumas. Así, utilizando la fórmula que nos brinda el teorema, podemos expresar la presencia de a como sigue:

$$|A| - S_1 + S_2 - S_3 + \cdots + (-1)^m S_m$$

$$= 1 - 0 + 0 - \cdots + (-1)^m(0) = \mathbf{1}$$

Lo que significa que cada elemento que no cumple con ninguna de las condiciones hace una aportación de 1 al lado izquierdo de la igualdad. A ese mismo elemento lo podemos ver como a uno de los que no son "eliminados".

Veamos ahora el caso en el que a cumple con exactamente n de las propiedades: Nuevamente, a aparece en A únicamente una vez, sin embargo, para el caso de S_1, aparecerá $\binom{n}{1}$ veces, pues a aparece en exactamente n subconjuntos (de tamaño 1); luego, en S_2 aparecerá $\binom{n}{2}$ veces, y así sucesivamente. Por tanto, podemos expresar la presencia de a en el lado derecho de la ecuación como:

$$|A| - S_1 + S_2 - S_3 + \cdots + (-1)^m S_m$$

$$= 1 - \binom{n}{1} + \binom{n}{2} - \cdots + (-1)^m \binom{n}{m}$$

$$= \binom{n}{0} - \binom{n}{1} + \binom{n}{2} - \cdots + (-1)^n \binom{n}{n} \tag{1.2}$$

Por la proposición 3, sabemos que:

$$\binom{n}{0} + \binom{n}{2} + \binom{n}{4} + \cdots = \binom{n}{1} + \binom{n}{3} + \binom{n}{5} + \cdots$$

Por lo que la ecuación 1.2 es igual a $\mathbf{0}$, lo que implica que un elemento que cumple al menos una de las propiedades no hace ninguna contribución a $\left|\bigcap_{i=1}^{m} \overline{A_i}\right|$, es decir,

fue "tachado".

Resumiendo lo que hicimos a lo largo de la demostración, probamos que cuando un objeto no cumple con ninguna de las P_i propiedades, éste se contará dentro de $\bigcap_{i=1}^{m} \overline{A_i}$ (los objetos que no cumplen con ninguna propiedad); en caso contrario, dicho objeto no se contará, es decir, no estará en $\bigcap_{i=1}^{m} \overline{A_i}$.

□

Ejemplo 38. Considera la frase "VEN AQUÍ HOY". Si eliminamos los espacios, ¿cuántas cadenas podemos formar tales que en ninguna de ellas aparezca ninguna de las tres palabras que originalmente aparecen en la frase?

Solución. Consideremos a P_1 como la propiedad de que en alguna cadena aparezca la palabra "VEN", a P_2 de que esté la palabra "AQUÍ" y a P_3 de que encontremos a la palabra "HOY"; sea A_i el subconjunto de cadenas que cumplen con la propiedad P_i.

Con lo que hemos aprendido a lo largo de este capítulo y viendo que disponemos de 10 letras distintas, es fácil ver que existen 10! permutaciones de la frase dada (sin espacios), es decir, $|A| = 10!$ Concentrémonos entonces en contar las cadenas que cumplen con alguna de las propiedades mencionadas antes.

Para contar las cadenas que incluyen a la palabra "VEN", veamos a toda la palabra como un sólo caracter. Así, tendremos 8 caracteres que permutar, lo que se puede hacer de 8! formas. Análogamente, las cadenas con las palabras "AQUÍ" y "HOY", son 7! y 8!, respectivamente.

Ahora contemos las cadenas que tienen al menos dos de las propiedades simultáneamente. Usaremos la misma lógica del caso anterior, viendo a cada palabra como un sólo caracter, así, $|A_1 \cap A_2| = 5!$, (y contará las cadenas en las que se encuentran las palabras "VEN" y "AQUÍ"), $|A_1 \cap A_3| = 6!$ y $|A_2 \cap A_3| = 5!$.

Finalmente, contamos las cadenas con las tres propiedades: $|A_1 \cap A_2 \cap A_3| = 3!$, y aplicamos el Principio de Inclusión y Exclusión.

$$\left|\bigcap_{i=1}^{m} \overline{A_i}\right| = 10! - (8! + 7! + 8!) + (5! + 6! + 5!) - 3!$$

Ejemplo 39. ¿Cuántos enteros positivos menores que algún natural n existen que sean *primos relativos*[2] con n?

Solución. Por el *Principio Fundamental de Aritmética*[3], sabemos que podemos escribir a n como un producto único de primos:

$$n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \cdots \cdot p_k^{\alpha_k}$$

Inicialmente tenemos un conjunto de n números naturales (todos los menores o iguales que n). Si eliminamos todos los enteros que tengan como factor a alguno de los primos que descomponen a n, estaremos contando a los primos relativos del mismo.

Sea P_i la propiedad de que un número sea divisible por p_i, y sea A_i el subconjunto de elementos que cumplen con la propiedad P_i. Ya que el conjunto que estamos buscando es $|\overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_k}|$, fijémonos en lo siguiente:

$$S_1 = \frac{n}{p_1} + \frac{n}{p_2} + \cdots + \frac{n}{p_k}$$

$$S_2 = \frac{n}{p_1 p_2} + \frac{n}{p_1 p_3} + \cdots + \frac{n}{p_{k-1} p_k}$$

$$\vdots$$

$$S_k = \frac{n}{p_1 p_2 \ldots p_k}$$

Aplicando en Principio de Inclusión y Exclusión, la cantidad de primos relativos con n menores que él es:

[2] Dos enteros son primos relativos si no tienen factores en común distintos de 1 y -1. En el capítulo de *Teoría de Números* se explica con mayor profundidad, en la sección de *Máximo Común Divisor*.
[3] Se explica en el capítulo de *Teoría de Números*, en la sección de *Primos*.

$$\left|\bigcap_{i=1}^{m} \overline{A_i}\right| = |A| - S_1 + S_2 - \cdots + (-1)^k S_k$$

$$= n - \left(\frac{n}{p_1} + \cdots + \frac{n}{p_k}\right) + \left(\frac{n}{p_1 p_2} + \cdots + \frac{n}{p_{k-1} p_k}\right) - \cdots + (-1)^k \left(\frac{n}{p_1 p_2 \ldots p_k}\right)$$

$$= n \left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_k}\right)$$

La expresión a la que llegamos es matemáticamente correcta, sin embargo, podemos tener problemas de precisión al realizar las divisiones de $1/p_i$, por lo que optaremos por programar la siguiente equivalencia:

$$n \left(\frac{p_1 - 1}{p_1}\right)\left(\frac{p_2 - 1}{p_2}\right) \cdots \left(\frac{p_k - 1}{p_k}\right)$$

Al número de enteros positivos que son primos relativos con n menores que él, es decir, a la expresión que acabamos de calcular, se le conoce como *Función de Euler*, y suele denotarse con $\varphi(n)$; es una generalización del Pequeño Teorema de Fermat.

Para la codificación de la solución invocaremos a un arreglo de primos que suponemos ya haber obtenido; este procedimiento lo puedes realizar de manera eficiente haciendo uso de la *Criba de Eratóstenes*, que podrás encontrar en la sección de *Primos* del capítulo de *Teoría de Números*.

```
1  int primos_relativos()
2  {
3      int n, i = 0, respuesta, factor_primo = primos[i];
4      cin>>n;
5      respuesta = n;
6
7      while (factor_primo * factor_primo <= n)
8      {
9          if (n % factor_primo == 0)
10             respuesta = respuesta * (factor_primo - 1) / factor_primo;
11
```

```
12          //Eliminamos todos los primos p_i de n.
13          while (n % factor_primo == 0)
14              n = n / factor_primo;
15
16          i = i + 1
17          factor_primo = primos[i];
18      }
19
20      //Si después de considerar todos los primos del arreglo, n es
21      //mayor que 1, implicamos que n es un primo mayor que los del
22      //arreglo, y lo incluimos en la solución.
23      if (n != 1)
24          respuesta = respuesta * (n - 1) / n;
25
26      return respuesta;
27  }
```

Ejemplo 40. Esta vez tendremos una fila de n niños que acaban de ingresar a la escuela primaria. Para favorecer la convivencia entre todos los estudiantes, los profesores decidieron formar a los niños en una fila el primer día de clases, y cada día repiten el proceso con la única condición de que ningún estudiante esté detrás de la misma persona que el primer día. ¿Cuántas filas distintas se pueden formar?

Solución. Numeremos a los niños de 1 a n respetando el orden de la fila del primer día. Luego, para el resto de los días se debe cumplir que en la fila no se encuentren juntos los niños 1 y 2, 2 y 3, y así sucesivamente hasta $n-1$ y n. Definamos entonces como P_i al hecho de que en alguna fila aparezcan juntos los niños i e $i+1$, para $1 \leq i \leq n-1$, y como A_i al subconjunto de filas que cumplen la propiedad P_i.

Consideremos a cada pareja no válida de niños como un sólo elemento; con esto tendremos $n-1$ objetos. Pensemos primero en las filas en las que pareja P_1 aparece para contar las filas que pertenecen a A_1; así, tendremos $n-1$ elementos para acomodar $(\{12, 3, 4, \ldots, n\})$, lo que se puede realizar de $(n-1)!$ formas. Luego, ya que existen $n-1$ parejas, $S_1 = (n-1)(n-2)!$.

Cuando se cumplen dos de las $n-1$ propiedades, existen dos posibilidades: que las parejas no compartan un niño o que lo hagan. En el primer caso, cada pareja contará como un sólo elemento, por lo que dispondre-

mos de $n-2$ objetos para acomodar; en el segundo caso, podemos ver a las dos parejas como un trío, lo que nos llevará a contar $n-2$ elementos también. Además, como existen $\binom{n-1}{2}$ posibles pares de propiedades, $S_2 = \binom{n-1}{2}(n-2)!$.

En general, en una fila en la que cohabitan k propiedades, existen $\binom{n-1}{k}$ subconjuntos de P_i's y debemos permutar únicamente $n-k$ objetos, por lo que $S_k = \binom{n-1}{k}(n-k)!$.

Aplicando el principio de inclusión y exclusión y tomando en cuenta que hay $n!$ filas posibles sin ninguna restricción:

$$\left|\bigcap_{i=1}^{n-1} \overline{A_i}\right| = |A| - S_1 + S_2 - \cdots + (-1)^{n-1} S_{n-1}$$

$$= n! - \binom{n-1}{1}(n-1)! + \binom{n-1}{2}(n-2)! - \cdots + (-1)^{n-1}\binom{n-1}{n-1}1!$$

Desarrollando la expresión $\binom{n-1}{i}(n-i)!$:

$$\binom{n-1}{i}(n-i)! = \frac{(n-1)!}{i!(n-1-i)!} \cdot (n-i)!$$

$$= (n-1)! \cdot \frac{(n-i)!}{i!(n-1-i)!}$$

$$= (n-1)! \cdot \frac{(n-i)\cancel{(n-i-1)!}}{i!\cancel{(n-i-1)!}}$$

$$= (n-1)! \cdot \frac{n-i}{i!}$$

Sustituyendo en la suma obtenida con el principio de inclusión y exclusión:

$$\left|\bigcap_{i=1}^{n} \overline{A_i}\right| = (n-1)!\left[\frac{n}{0!} - \frac{n-1}{1!} + \frac{n-2}{2!} - \cdots + (-1)^{n-1}\frac{1}{(n-1)!}\right]$$

```
int solucion()
{
    int i, n, respuesta = 0, signo;
    cin>>n;

    for (i = 0; i < n; i = i + 1)
    {
        signo = 1;
        if (i % 2)
            signo = -1;

        respuesta = respuesta + signo * factorial(n - 1) * (n - i)
                  / factorial(i)
    }

    return respuesta;
}
```

Ejemplo 41. Imagina que tienes una tarjeta de cierto banco, pero olvidas el NIP. Al hacer los trámtites necesarios para la recuperación del número, te muestran el NIP anterior y te piden que crees un nuevo NIP, que tiene que contener los mismos caracteres que el anterior, pero con la condición de que la nueva cadena no contenga caracteres contiguos que en el NIP olvidado hayan estado juntos también.

¿Cuántas cadenas se están excluyendo al exigir esa condición si se sabe que no puede haber dos caracteres iguales en un NIP?

> **Solución.** Ya que sabemos que el NIP olvidado se compone de n caracteres y que ninguno de ellos aparece más de una vez, sabemos que el nuevo NIP se compondrá de exactamente n caracteres distintos también.
>
> Llamemos F al NIP anterior y N al NIP que debe crearse; nombremos como P_i a la condición de que suceda que los elementos $F[i]$ y $F[i+1]$ (para $1 \leq i \leq n-1$) sean contiguos en N, y sea A_i el conjunto de cadenas que cumplen con P_i.
>
> Nota que este problema se parece mucho al anterior, con la diferencia de que lo que buscamos aquí es el complemento de lo que resulta de aplicar el principio de inclusión y exclusión, es decir, el número de cadenas

que cumplen con una (o más) propiedades.

Comencemos por contar los N's que cumplen con al menos una P_i. Dado que F se compone de n elementos, se cuentan $n-1$ parejas de caracteres contiguos. Partiremos de este hecho para analizar los posibles nuevos NIP's: Tomemos alguna de las parejas, digamos P_t y veámosla como un único elemento (esto implica que sus caracteres no serán separables). De esta forma, nos quedarán $n-1$ elementos para colocar en el nuevo NIP y $n-1$ lugares para hacerlo, esto es $(n-1)!$; ya que este proceso se puede hacer para $n-1$ pares de caracteres, las cadenas que cumplen con al menos una condición son $(n-1)(n-1)!$.

En general, como vimos en el ejemplo anterior, para k condiciones que cohabitan en la misma cadena, se cumple que habrá $n-k$ elementos que acomodar en $n-k$ lugares y se tendrán $\binom{n-1}{k}$ subconjuntos de k propiedades, es decir, $\binom{n-1}{k}(n-k)!$ NIP's distintos.

Ya que deseamos obtener el complemento de las cadenas que NO cumplen con ninguna condición, la operación que resuelve el problema obedece al siguiente procedimiento:

$$\left|\bigcup_{i=1}^{n-1}\overline{A_i}\right| = |A| - \left[|A| - S_1 + S_2 - \cdots + (-1)^{n-1}S_{n-1}\right]$$

$$= S_1 - S_2 + \cdots + (-1)^{n-1}S_{n-1}$$

$$= \binom{n-1}{1}(n-1)! - \binom{n-1}{2}(n-2)! + \cdots + (-1)^{n-1}\binom{n-1}{n-1}1!$$

$$= \frac{(n-1)!}{1!\cdot(n-2)!}(n-1)! - \frac{(n-1)!}{2!\cdot(n-3)!}(n-2)! + \cdots + (-1)^{n-1}\frac{(n-1)!}{(n-1)!\cdot 0!}1!$$

$$= (n-1)!\left[\frac{n-1}{1!} - \frac{n-2}{2!} + \cdots + (-1)^{n-1}\frac{1}{(n-1)!}\right]$$

```
int solucion()
{
    int n, i, respuesta = 0, signo;
    cin>>n;

    for (i = 1; i < n; i = i + 1)
    {
        signo = -1;
        if (i % 2)
            signo = 1;

        respuesta = respuesta + signo * factorial(n - 1) * (n - i)
                  / factorial(i);
    }

    return respuesta;
}
```

Nota que tanto en el programa del problema 40 como en el de este ejemplo, pese a que es posible factorizar la expresión matemática a la que llegamos (sacando $(n-1)!$), cuando la solución es llevada al código, no se considera dicha factorización. Esto debido a la baja precisión de la computadora para conservar todos los decimales de números no enteros.

Ejemplo 42. Entras a una tienda con la intención de comprar n galletas de 3 sabores diferentes. En la entrada de este problema recibirás, después de n, tres enteros no negativos a_1, a_2, a_3, que indicarán que debes llevar al menos a_i galletas del sabor i. Luego recibirás otros tres enteros no negativos b_1, b_2, b_3, tales que $a_i \leq b_i$, que querrán decir que existen únicamente b_i galletas del sabor i disponibles en la tienda.

Considera que para toda entrada se cumplirá que $a_1 + a_2 + a_3 \leq n \leq b_1 + b_2 + b_3$.

Solución. Sean x_1, x_2 y x_3 el número de galletas que llevarás del sabor a_1, a_2 y a_3, respectivamente. Por comodidad, nombremos como A a la suma de las tres a_i's y como B a la suma de las tres b_i's.

Ya que en total pretendes llevar n galletas, podemos resumir lo que describe el problema con la siguiente expresión:

$$x_1 + x_2 + x_3 = n \tag{1.3}$$

$$a_i \leq x_i \leq b_i,\ i = 1, 2, 3$$

Introducimos una nueva variable $y_i = x_i - a_i$, de la que deducimos que $x_i = y_i + a_i$, con lo que podemos reescribir la ecuación 1.3 como

$$y_1 + y_2 + y_3 = n - A \tag{1.4}$$

$$0 \leq y_i \leq b_i - a_i,\ i = 1, 2, 3$$

Ya que sabemos que en la entrada siempre se cumple que $A \leq n$, podemos asegurar que $n - A$ es un entero no negativo. Además, construimos restricciones en las que las y_i también toman valores de únicamente enteros no negativos. Así, vemos que éste es un problema muy similar al del ejemplo 29, resuelto con *separadores*, con la diferencia de que en este ejercicio existe un límite extra para los valores de los sumandos, y es aquí donde el principio de inclusión y exclusión nos será de utilidad.

Llamamos P_i a la propiedad de que $y_i \geq b_i - a_i + 1$, y nuevamente, A_i al subconjunto de tuplas en las que se cumpla P_i. Con esto, buscamos hallar el número de ternas $< y_1, y_2, y_3 >$ que, sumadas, resultan $n - A$, y que no cumplan con ninguna de las propiedades P_i.

La primera pregunta que debemos responder es cuántas tuplas sin considerar ninguna restricción P_i existen. Siguiendo el razonamiento del ejemplo 29, tenemos $n - A$ objetos a los que debemos restar 3 para asegurar que todos los y_i sean naturales; después sumamos los dos separadores que requerimos. Eligiendo esos separadores, obtenemos $\binom{n-A-1}{2}$ ternas.

Analicemos ahora qué sucede cuando aseguramos que una terna cumpla con al menos una de las propiedades. Sin pérdida de generalidad, pensemos en P_1. Ya que pretendemos asegurar que $y_1 > b_1 - a_1$, tenemos solamente $n - A + 2 - (b_1 - a_1 + 1)$ objetos para repartir entre las tres variables, y no importará la forma en que lo hagamos, ya nos aseguramos de que toda terna obtenida cumplirá con P_1. Eligiendo los dos separadores requeridos, llegamos a la expresión:

$$\binom{n-A+2-(b_1-a_1+1)}{2}$$

Y luego consideramos las tres propiedades existentes.

$$S_1 = \binom{n-A+2-(b_1-a_1+1)}{2} + \binom{n-A+2-(b_2-a_2+1)}{2} + \binom{n-A+2-(b_3-a_3+1)}{2}$$

Pensemos ahora en la pareja P_1 y P_2. Nos aseguramos de que ambas propiedades se cumplan para una terna y obtenemos la expresión:

$$\binom{n-A+2-(b_1-a_1+1)-(b_2-a_2+1)}{2}$$

Tomando en cuenta los 3 pares de propiedades que existen:

$$\begin{aligned} S_2 = &\binom{n-A+2-(b_1-a_1+1)-(b_2-a_2+1)}{2} \\ &+ \binom{n-A+2-(b_1-a_1+1)-(b_3-a_3+1)}{2} \\ &+ \binom{n-A+2-(b_2-a_2+1)-(b_3-a_3+1)}{2} \end{aligned}$$

Finalmente obtenemos S_3.

$$S_3 = \binom{n-A+2-(b_1-a_1+1)-(b_2-a_2+1)-(b_3-a_3+1)}{2}$$

Aplicamos el principio de inclusión y exclusión y el problema se termina, sin embargo hay una cosa más que debemos considerar antes de hacerlo: En la expresión $\binom{n}{k}$, por definición, se debe cumplir que $k \leq n$. Como no podemos asegurar que eso pase en los coeficientes que hemos obtenido en

la solución de este problema, debemos incluir en el algoritmo una restricción que dicte que si $k > n$, $\binom{n}{k} = 0$, lo que, en términos de combinatoria, significa que no existe ningún subconjunto de tamaño k de un conjunto de tamaño n si k es mayor que n.

```
int solucion()
{
    int i, j n, a[4], b[4], A, B, S0 = 0, S1 = 0, S2 = 0, S3 = 0;
    cin>>n>>a[1]>>a[2]>>a[3]>>b[1]>>b[2]>>b[3];
    A = a[1] + a[2] + a[3];

    if (n - A - 1 >= 2)
        S0 = binomial(n - A - 1, 2);

    for (i = 1; i <=3; i = i + 1)
        if (n - A + 2 - (b[i] - a[i] + 1) >= 2)
            S1 = S1 + binomial(n - A + 2 - (b[i] - a[i] + 1), 2);

    for (i = 1; i <= 2 i = i + 1)
        for (j = i + 1; j <= 3; j = j + 1)
            if (n - A + 2 - (b[i] - a[i] + 1) - (b[j] - a[j] + 1) >= 2)
                S2 = S2 + binomial(n - A + 2 - (b[i] - a[i] + 1)
                   - (b[j] - a[j] + 1), 2);

    if (n-A+2 - (b[1]-a[1]+1) - (b[2]-a[2]+1) - (b[3]-a[3]+1) >= 2)
        S3 = binomial(n - A + 2 - (b[1] - a[1] + 1) - (b[2] -
            a[2] + 1) - (b[3] - a[3] + 1), 2);

    return S0 - S1 + S2 - S3;
}
```

1.7.1. Subfactorial

Ejemplo 43. Imagina una fila de n personas. A la primera de ellas le damos un boleto con el número 1, a la segunda con el número 2, y así sucesivamente hasta la n-ésima persona. Luego, le pedimos a todos que abandonen su lugar y que tomen cualquier otro en la fila con la única condición de que no vuelvan a su sitio original. ¿Cuántos acomodos distintos de la fila se pueden crear con estas condiciones?

Solución. Llamemos P_i a la propiedad de que la persona con el número i esté en el i-ésimo lugar, y A_i al subconjunto de filas que cumplen con la propiedad P_i. Como queremos saber en cuántos casos ninguna de las personas está en su lugar original, estamos buscando la cardinalidad del conjunto $\left|\bigcap_{i=1}^{n} \overline{A_i}\right|$.

Para aplicar el Principio de Inclusión y Exclusión es necesario primero conocer $|A|$, S_1, $S_2, \ldots, S_n$, así que comencemos por ahí.

Sin considerar ninguna restricción de las propiedades, sabemos que existen $n!$ permutaciones de las personas que se encuentran en la fila, o sea que $|A| = n!$.

Sea k algún número entre 1 y n, inclusive. Fijemos a la persona con el boleto k en el k-ésimo lugar. Como restan $n-1$ lugares, las $n-1$ personas que quedan se pueden acomodar de $(n-1)!$ formas, y dado que existen n valores que puede tomar k, y para todos se repetirá el mismo proceso, $S_1 = n(n-1)!$.

Luego fijemos a dos personas en su lugar original. Para el resto quedan $(n-2)!$ formas de acomodar a los demás individuos, y ya que existen $\binom{n}{2}$ pares distintos de personas que pueden ser fijadas, $S_2 = \binom{n}{2}(n-2)!$.

En general, si fijamos $j \leq n$ personas en su lugar original, restarán $n-j$ lugares y $n-j$ personas, que podrán ser colocadas de $(n-j)!$ maneras. Por otro lado, existen $\binom{n}{j}$ grupos de personas que pueden ser fijados, por lo tanto $S_j = \binom{n}{j}(n-j)!$.

Finalmente, podemos aplicar la fórmula que nos brinda el teorema 6.

$$\left|\bigcap_{i=1}^{n} \overline{A_i}\right| = n! - n(n-1)! + \binom{n}{2}(n-2)! - \cdots + (-1)^{n-1}\binom{n}{n-1}(1) + 1$$

$$= \sum_{i=0}^{n}(-1)^i \binom{n}{i}(n-i)!$$

$$= n! - \frac{n!}{1!\cancel{(n-1)!}}\cancel{(n-1)!} + \frac{n!}{2!\cancel{(n-2)!}}\cancel{(n-2)!} - \cdots + (-1)^n \frac{n!}{n! \cdot \cancel{0!}}\cancel{0!}$$

$$= n!\left[1 - \frac{1}{1!} + \frac{1}{2!} - \cdots + (-1)^n \frac{1}{n!}\right] \tag{1.5}$$

$$= n! - \frac{n!}{1!} + \frac{n!}{2!} - \cdots + (-1)^n \frac{n!}{n!} \tag{1.6}$$

```
int subfactorial()
{
    int n, i, respuesta = 0, signo;
    cin>>n;

    for (i = 0; i <= n; i = i + 1)
    {
        signo = 1;
        if (i % 2)
            signo = -1;

        respuesta = respuesta + signo * factorial(n) / factorial(i);
    }

    return respuesta;
}
```

Este ejemplo nos sirve para teorizar lo que se conoce como *subfactorial*: El subfactorial de un conjunto cuenta un tipo especial de permutaciones, en las que cada objeto tiene una posición inicial y no le es permitido volver a ella. En inglés, estos acomodos son conocidos como *derangements*, y aunque en español no hay una traducción exacta, aquí los nombraremos "desarreglos".

Nota que el subfactorial de los elementos de un conjunto nos brinda un caso particular del principio de inclusión y exclusión, en el que todos los A_i tienen la misma cardinalidad, al igual que las intersecciones de los pares de conjuntos $(A_i \cap A_j)$ (y, en general, todas las intersecciones de k A_i's), razón por la cual es posible simplificar la fórmula del teorema 6 como lo indica la expresión 1.5.

Por conveniencia y dada la falta de precisión de una computadora promedio al hacer divisiones no enteras, programamos la expresión 1.6, sin embargo, es muy común que en los textos de la materia te encuentres con la fórmula 1.5.

Ya que pasaremos un rato más hablando del subfactorial, usaremos una notación especial para referirnos a él: Llamaremos D_n al número de permutaciones distintas que se pueden hacer de un conjunto de cardinalidad n sin que ningún elemento esté en su posición original. Por ejemplo, en una fila de 3 personas numeradas, el subfactorial es 2, y los desarreglos válidos son $< 2, 3, 1 >$ y $< 3, 1, 2 >$.

Hemos visto ya una forma iterativa de obtener D_n. El siguiente teorema nos brinda una forma recursiva de llegar al mismo resultado.

Teorema 7. El número D_n satisface la siguiente relación:

$$D_n = \begin{cases} 0 & \text{si } n = 1 \\ 1 & \text{si } n = 2 \\ (n-1)(D_{n-2} + D_{n-1}) & \text{si } n \geq 3 \end{cases}$$

Demostración. Es fácil ver que si tenemos un conjunto de cardinalidad 1 no existe forma de acomodar el único objeto sin que éste regrese a su posición original, mientras que cuando $n = 2$, el único acomodo válido es $< 2, 1 >$. Veamos entonces lo que sucede cuando $n \geq 3$.

Consideremos todos los desarreglos del arreglo $< 1, 2, 3, \ldots, n >$. Aceptamos como válida una configuración si tiene la forma:

$$< a_1, a_2, \ldots, a_n > \text{ donde } a_i \neq i$$

Por su naturaleza, sabemos que a_1 puede tomar los valores $2, 3, \ldots n$, y que el número de acomodos válidos de todo el conjunto que comienzan con cada valor distinto de a_1 es el mismo. Sin pérdida de generalidad, digamos que $a_1 = 2$ y nombremos como d_2 al conjunto de desarreglos que comienzan con 2. Ya que a_1 puede tomar $n - 1$ valores distintos (de 2 a n), podemos asegurar que $D_n = (n-1)d_2$.

Lo que buscaremos ahora será expresar a d_2 en términos de algo conocido. Para el segundo elemento de nuestros desarreglos, tenemos dos opciones: que éste sea 1 o que sea cualquier otro número de los que restan.

Si $a_2 = 1$, nuestros arreglos válidos serán de la forma $< 2, 1, a_3, a_4, \ldots, a_n >$. Éste es un arreglo con las dos primeras posiciones definidas y para las $n-2$ posiciones restantes se debe cumplir que $a_i \neq i$, lo que podemos traducir como el subfactorial de $n-2$, es decir, D_{n-2}.

Por otro lado, si $a_2 \neq 1$, podemos ver a las permutaciones admisibles como aquéllas con la forma $< 2, a_2, a_3 \ldots, a_n >$, en donde $a_2 \neq 1$, $a_3 \neq 3$, $\ldots$,$a_n \neq n$, lo que puede ser expresado como D_{n-1}.

De lo anterior, concluimos que otra manera de expresar d_2 es como la suma de $D_{n-2} + D_{n-1}$.

Así, cuando $n \geq 3$, llegamos a la igualdad:

$$D_n = (n-1)(D_{n-2} + D_{n-1}) \tag{1.7}$$

□

Este teorema nos regala una relación recursiva en función de los dos términos previos. ¿Podremos hallar una expresión recursiva que sólo dependa del término inmediatamente anterior?

Es posible reescribir la ecuación 1.7 como sigue:

$$\begin{aligned} D_n &= (n-1)D_{n-2} + (n-1)D_{n-1} \\ &= (n-1)D_{n-2} + nD_{n-1} - D_{n-1} \end{aligned}$$

Restando nD_{n-1} de ambos lados:

$$\begin{aligned} D_n - nD_{n-1} &= -D_{n-1} + (n-1)D_{n-2} \\ D_n - nD_{n-1} &= -[D_{n-1} - (n-1)D_{n-2}] \end{aligned}$$

Nota que la expresión que está adentro de los corchetes es exactamente la misma que la que obtuvimos de lado izquierdo en la ecuación sustituyendo n por $n-1$. De aquí que podemos expresar la ecuación de la siguiente manera.

$$\begin{aligned} D_n - nD_{n-1} &= -[D_{n-1} + (n-1)D_{n-2}] \\ &= (-1)^2[D_{n-2} - (n-2)D_{n-3}] \\ &= \cdots \\ &= (-1)^{n-2}[D_2 - 2D_1] \end{aligned}$$

Del teorema 7 sabemos que $D_1 = 0$ y que $D_2 = 1$, por lo que podemos reescribir la última igualdad como:

$$D_n - nD_{n-1} = (-1)^{n-2}[1-0] = (-1)^{n-2}$$

Así, llegamos al hecho de que $D_n - nD_{n-1} = (-1)^{n-2}$, y ya que $n-2$ tiene la misma paridad que n, concluimos que:

$$D_n = nD_{n-1} + (-1)^n$$

.

Ejemplo 44. A una fiesta llegan n matrimonios que entregan sus abrigos en la recepción. A la salida hay una confusión y a todas las personas les entregan un abrigo que no les pertenece, pero eso sí, un abrigo de una persona de su género. ¿De cuántas formas distintas pudo haber sucedido esto?

Solución. Nota que la confusión entre los abrigos de hombres y mujeres son eventos independientes.

Ya que todos los abrigos de mujeres fueron entregados a mujeres y existe una persona "prohibida" para cada uno, se cumplen las características para contar los desarreglos, con el subfactorial de un conjunto de tamaño n, esto es D_n.

Como lo mismo sucede para el caso de los hombres, y para cada conteo de un sexo se deben contar todas las posibilidades del otro, llegamos

a la expresión D_n^2.

```
int D(int n)
{
    if (n == 1)
        return 0;
    return n * D(n - 1) + (n % 2 ? -1 : 1);
}

int solucion()
{
    int n;
    cin>>n;
    return D(n) * D(n);
}
```

Ejemplo 45. Sea R un arreglo de tamaño n cuyos índices comienzan en 0. Si deseamos meter en cada posición del arreglo un entero distinto perteneciente al conjunto $\{0, 1, 2, \ldots, n-1\}$, ¿de cuántas formas es posible asignar los n números si queremos asegurar que para las últimas k posiciones se cumpla que $R[i] \neq i$?

Considera a n y a k como las entradas del problema.

Solución. Fijemos nuestra atención en las últimas k posiciones del arreglo. Llamaremos P_i a la propiedad de que $R[i] = i$, para $n-k \leq i \leq n-1$, y como es costumbre, nombraremos A_i al conjunto de arreglos que cumplen con la propiedad P_i.

Si fijamos el número i en la posición $R[i]$, perteneciente a las últimas k posiciones del arreglo, podemos acomodar los enteros restantes de $(n-1)!$ formas, y como existen k propiedades, $S_1 = k(n-1)!$.

Si nos aseguramos de que un arreglo cumpla con al menos dos propiedades, tendremos dos elementos definidos y $(n-2)!$ maneras de acomodar los restantes. Ya que existen $\binom{k}{2}$ parejas posibles en las últimas k posiciones, $S_2 = \binom{k}{2}(n-2)!$.

En general, si un arreglo tiene j elementos en el índice que tienen prohibi-

do, los enteros restantes pueden ser situados de $(n-j)!$ modos, y ya que existen $\binom{k}{j}$ posibles conjuntos de tamaño j en las últimas k posiciones, $S_j = \binom{k}{j}(n-j)!$.

Por último, buscando el conjunto de arreglos que no cumplen con ninguna propiedad, aplicamos el principio de inclusión y exclusión:

$$\left|\bigcap_{i=n-k}^{n-1} \overline{A_i}\right| = n! - \binom{k}{1}(n-1)! + \binom{k}{2}(n-2)! - \cdots + (-1)^k\binom{k}{k}(n-k)!$$

$$= \sum_{i=0}^{k}(-1)^i\binom{k}{i}(n-i)!$$

```
int solucion()
{
    int i, n, k, respuesta = 0;
    cin>>n>>k;

    for (i = 0; i <= k; i = i + 1)
    {
        int signo = 1;
        if (i % 2)
            signo = -1;

        respuesta = respuesta + signo * binomial(k,i) * factorial(n-i);
    }

    return respuesta;
}
```

2

Teoría de Números

En la teoría de números descansan las bases de varios algoritmos que permiten resolver ágilmente diversos problemas. Por ejemplo, es común encontrar problemas que requieran el uso de datos tan grandes que no es posible almacenarlos tal cual, y en vez de ello, los representaremos a través de su equivalente módulo n; la aritmética modular es uno de los temas que aquí se abordan, pero para estudiarla y comprenderla, es necesario entender primero alguna propiedades básicas de divisibilidad. Lo anterior nos permitirá aprender técnicas más complicadas, como lo que llamamos *Rolling Hash* y *Factorial Reverso*. Este capítulo revisará varios temas más, que comprenden algunas propiedades de primos, los criterios de divisibilidad más utilizados, una técnica que permite obtener la potencia de un número de manera logarítmica y el máximo común divisor de varios enteros que, a su vez, será de ayuda para llegar al mínimo común múltiplo de los mismos.

Comenzaremos por algunas definiciones sencillas e iremos aumentando la complejidad de los temas. En caso de que tengas algún problema con los temas básicos, puedes consultar [8], en donde dispondrás de una gran cantidad de ejemplos, aunque si buscas un material con mayor formalidad, [9] es una buena opción para comenzar.

2.1. Divisibilidad: Algunas propiedades

Establezcamos primero la terminología necesaria.

Sean a, b dos números enteros. Decimos que b es *múltiplo* de a si se cumple que $ac = b$ para algún entero c. Inversamente, llamamos a a y a c *factores* o *divisores* de b, pues ambos lo dividen produciendo residuo cero.

La simbología utilizada para tratar con temas de divisibilidad es $a \mid b$, y se lee como "a divide a b".

Una vez establecido lo anterior, veremos algunas propiedades de divisibilidad de los enteros, que si bien pueden ser deducidas de manera intuitiva, es necesario desarrollar sus demostraciones con el fin de comprender la razón de que se cumplan los resultados que revisaremos más adelante.

Proposición 5. Sean $a, b, c \in \mathbb{Z}$. Entonces:

1. $1 \mid a$.
2. $a \mid a$ para $a \neq 0$.
3. Si $a \mid b$ y $b \mid c$, entonces $a \mid c$ para $a, b \neq 0$.
4. Si $a \mid b$ y $b \mid a$, entonces $|a| = |b|$ para $a, b \neq 0$.
5. Si $a \mid b$, entonces $a \mid bc$ para todo entero c y para $a \neq 0$.
6. Si $a \mid b$ y $a \mid c$, entonces $a \mid (b + c)$ para $a \neq 0$.

Demostración. Los primeros dos puntos resultan directamente del hecho de que cualquier número a se puede escribir como el producto de $1 \times a$, así que nos concentraremos en demostrar el resto de las propiedades:

3. Para demostrar la propiedad transitiva, veamos que, ya que $a \mid b$, podemos expresar a b como $b = a \cdot a_1$, para algún entero a_1; de la misma forma, podemos escribir a c como $c = b \cdot b_1$, para algún entero b_1; a su vez, c es igual a $c = a \cdot a_1 \cdot b_1$, con lo que podemos decir que c es múltiplo de a, y por tanto, que $a \mid c$.

4. Para que $a\big|b$, se tiene que cumplir que $|a| \leq |b|$, y para que $b\big|a$, es necesario que $|b| \leq |a|$. La única manera de que ambas relaciones se cumplan es que $|a| = |b|$.

5. Sea a_1 un entero tal que $b = a \cdot a_1$. El producto $b \cdot c$ se puede escribir como $b \cdot c = a \cdot a_1 \cdot c$, por lo tanto, $a\big|bc$.

6. Expresemos a b y a c como $b = a \cdot a_1$ y $c = a \cdot a_2$. De aquí, $b+c = (a \cdot a_1)+(a \cdot a_2) = a(a_1 + a_2)$. Por tanto, $a\big|(b + c)$.
   Nota que si b o c tienen signo negativo, estamos probando que esta propiedad se extiende también a la resta, es decir, $a\big|(b - c)$.

Como puedes ver, en muchas ocasiones, cuando estamos intentado demostrar alguna propiedad de divisibilidad, puede ser conveniente escribir al dividendo como el producto del divisor por algún entero x.

□

Acabamos de probar las bases elementales de la divisibilidad. Procedamos ahora con resultados un poco más elaborados.

Proposición 6. Sean a, b, c números enteros. Se cumple que:

1. $a\big|b$ si y sólo si $ac\big|bc$, para a, c distintos de 0.

2. Si $a\big|b$ y $a\big|c$, entonces $a\big|(bm + cn)$ para $a \neq 0$ y cualesquiera m y n enteros.

3. Sea p un número primo. Si $p\big|ab$, entonces $p\big|a$ y/o $p\big|b$.

4. Sea p un primo. Si $p\big|ab$ y a y p no tienen factores en común, entonces $p\big|b$.

Demostración.

1. Probaremos primero que si $a\big|b$, se cumplirá que $ac\big|bc$: Ya que $a\big|b$, $b = a \cdot a_1$, para algún $a_1 \in \mathbb{Z}$. De ahí, $b \cdot c = a \cdot a_1 \cdot c$, y por lo tanto, $ac\big|bc$.
   Procediendo con el recíproco: Como $ac\big|bc$, es posible escribir a bc como $bc = ac \cdot k$, $k \in \mathbb{Z}$. Se sigue que $b = a \cdot k$, por lo que $a\big|b$.

2. Del punto 5 de la proposición 5, sabemos si que $a|b$ y $a\Big|c$, podemos decir que $a\Big|bm$ y $a\Big|cn$, y del punto 6 concluimos que es cierto que $a\Big|(bm+cn)$.

3. Ya que $p\Big|ab$, debe existir un entero k tal que $ab = pk$, de lo que se deduce que $k = (\frac{a}{p})(b)$, o bien, que $k = (a)(\frac{b}{p})$. Como estamos seguros de que k es un entero, se tiene que cumplir que, o $p\Big|a$ o $p\Big|b$, e incluso, pueden ocurrir ambos casos simultáneamente.

4. Directa del punto anterior, considerando que ya que a y p no tienen factores en común, se cumple que $p \nmid a$, y por tanto tiene que pasar que $p\Big|b$.

Y se concluye la demostración.

□

Analicemos ahora algunos resultados que involucran sumandos elevados a alguna potencia.

Proposición 7. Sean a y b dos números enteros y sea n un número natural.

1. $a - b\Big|a^n - b^n$.

2. $a + b\Big|a^n + b^n$ si n es un número impar.

Demostración.

1. Concentrémonos en la expresión $a^n - b^n$. Ésta es una diferencia de dos números cuyas potencias son iguales, lo que implica que puede ser factorizado de la siguiente manera:

$$a^n - b^n = (a-b)(a^{n-1} + a^{n-2}b + a^{n-3}b^2 + \cdots + ab^{n-2} + b^{n-1})$$

   De aquí, sabemos que $a - b$ es un factor de $a^n - b^n$.

2. En el segundo caso, ya que n es impar, podemos escribir a b^n como $b^n = -(-b)^n$, por lo que es posible transformar la expresión que queremos probar en:

$$a - (-b)\Big|a^n - (-b)^n$$

A la derecha tenemos, nuevamente, una diferencia de números cuyas potencias son iguales, por lo que se puede reescribir como:

$$a^n-(-b)^n=\Big[a-(-b)\Big]\Big[a^{n-1}+a^{n-2}(-b)+a^{n-3}(-b)^2+\cdots+a(-b)^{n-2}+(-b)^{n-1}\Big]$$

Con lo que concluimos que $a-(-b)=a+b$ divide a a^n+b^n.

Con lo anterior, terminamos la demostración.

□

Ejemplo 46. Desarrollando un programa que realiza una tarea que por ahora no nos compete, lees un conjunto de más de 2 cadenas de caracteres, mismas con la que has tenido problemas. El conflicto radica en que, pese a que has especificado al usuario que cada cadena debe contener exactamente $k\cdot z$, $z\in\mathbb{N}$, caracteres, suele no respetar esta regla.

Para analizar tu problema, creaste un subproceso que devuelve el tamaño de cada cadena y lo guarda en un arreglo, y para que tu programa original funcione, tienes que asegurar que, para cualquier par y terna de cadenas consecutivas que tomes, se cumplirá que suman un número de caracteres múltiplo de k, por lo que eliminarás a las cadenas que no cumplan con dicha condición.

Encuentra cómo suprimir esas cadenas sin hacer una búsqueda completa.

Solución. Nombremos T al arreglo que guarda el tamaño de las cadenas ingresadas. Salvo el primer y el último elemento de T, todos los números son parte de exactamente dos pares de elementos contiguos.

Tomemos entonces alguna terna $T[i-1]$, $T[i]$, $T[i+1]$, para $1\leq i\leq n-2$, en donde n es la cantidad de cadenas recibidas. En dicha terna, se tienen dos pares de números contiguos para los que se debe cumplir que $k\Big|T[i-1]+T[i]$ y que $k\Big|T[i]+T[i+1]$. Bifurquemos este planteamiento en dos escenarios: que $T[i]$ sea múltiplo de k y que no lo sea.

En el primer caso, para que se cumpla que $k\Big|T[i-1]+T[i]$, es necesario que $T[i-1]$ sea múltiplo de k, y para que $k\Big|T[i]+T[i+1]$, debe pasar que $T[i+1]$ sea múltiplo de k. Así, $k\Big|T[i-1]$, $k\Big|T[i]$ y $k\Big|T[i+1]$, y la suma

de la terna $T[i-1]$, $T[i]$, $T[i+1]$ también cumplirá la condición requerida.

Veamos la otra cara de la moneda y supongamos que k no divide a $T[i]$. En ese caso, para que se cumpla que $k \mid T[i-1] + T[i]$, ocurrirá que k no divide a $T[i-1]$, y para que $k \mid T[i] + T[i+1]$, pasará que k no divide a $T[i+1]$ (aclaremos que el hecho de que digamos que $k \nmid T[i-1]$ y que $k \nmid T[i+1]$ no implica que cualquier número que no sea múltiplo de k será útil, no obstante, para nuestros fines no requerimos más especificidad). Luego, podemos escribir la suma de la terna elegida como:

$$T[i-1] + T[i] + T[i+1] = (T[i-1] + T[i]) + (T[i] + T[i+1]) - T[i]$$

Podemos asegurar que $k \mid T[i-1] + T[i]$ y que $k \mid T[i] + T[i+1]$, por lo que la suma de ambas parejas también será divisible por k, no obstante, $k \nmid T[i]$, por lo que, al restarlo, nos quedamos con un número no divisible por k.

Así, concluimos que la única forma de que se cumpla lo pedido es que todas las cadenas tengan un número de elementos múltiplos de k, lo que significa que habrá que eliminar todas las cadenas cuyo tamaño no sea divisible por k, o lo que es lo mismo en este caso, no guardarlas.

```
vector<string> solucion()
{
    vector<string> guardando_cadenas;
    string cadena;
    int k;
    cin>>k;

    while (cin>>cadena)
    {
        cin>>cadena;
        if (cadena.size() % k == 0)
            guardando_cadenas.push_back(cadena);
    }

    return guardando_cadenas;
}
```

Ejemplo 47. Se tiene una cuadrícula de dimensiones $(n-1) \times (n+1)$, para un natural dado n mayor o igual que 5. Requieres asegurarte de que sea posible dividir dicha cuadrícula en exactamente 24 particiones de la misma área, con la condición de que está prohibido dividir un cuadro en 2 o más pedazos. (Problema modificado, tomado de [10]).

Solución. Primero, vamos a recordar que una *partición* es la división de un todo tal que la unión de las partes sea el mismo todo y la intersección entre cualquier par de ellas sea nula. Dicho esto, veamos lo necesario para que la partición se pueda llevar a cabo.

Nota que el área de la cuadrícula es de $(n-1)(n+1) = n^2 - 1$, y esto nos lleva a buscar cuándo $24 \Big| (n-1)(n+1)$. Hay que señalar que esta condición nos resulta suficiente porque no nos interesa la forma que tomen las particiones. Por otro lado, $n-1$ y $n+1$ son números naturales con una diferencia de 2. De aquí, podemos decir que ambos tienen la misma paridad, así que dividiremos el problema en dos casos: cuando ambos son impares y cuando ambos son pares.

Si $n-1$ y $n+1$ son impares, decimos que $2 \not| n-1$ y $2 \not| n+1$. Ahora, $24 = 8 \times 3$, lo que implica que si queremos que 24 divida a $n^2 - 1$, se debe cumplir que 8 divida, ya sea a $n-1$, o bien, a $n+1$; pero si ambos son impares, ninguna de las dos cosas va a ocurrir. Concluimos entonces que $n-1$ y $n+1$ no pueden ser impares, o sea, para que la partición sea posible, n no debe ser par.

Analicemos qué pasa si $2 \Big| n-1$ y $2 \Big| n+1$. Cada 4 números enteros hay un múltiplo de 4, o lo que es lo mismo, cada dos números pares. Ya que tenemos dos números pares consecutivos, podemos asegurar que uno de ellos es múltiplo de 4, y los podemos escribir como $2k_1$ y $4k_2$, para $k_1, k_2 \in \mathbb{N}$. Multiplicándolos, obtendremos $8 \cdot k_1 \cdot k_2$, lo que nos ayuda a deducir que $8 \Big| n^2 - 1$.

Lo único que falta es que alguno de los dos números $(n-1, n+1)$ sea múltiplo de 3. Para esto, hay que ver que los enteros $n-1$, n y $n+1$ son consecutivos, y ya que son 3, podemos asegurar que entre ellos hay exac-

tamente un múltiplo de 3; esto nos lleva a que si queremos que $3 \mid n^2 - 1$, requerimos que $3 \nmid n$, de esa forma el múltiplo de 3 se encontrará en $\{n-1, n+1\}$.

Resumiendo, la manera de asegurar que $24 \mid n^2 - 1$ es teniendo la certeza de que $2 \nmid n$ y $3 \nmid n$.

```
bool solucion()
{
    int n;
    bool particion = false; //True si es posible.
    cin>>n;

    if (n % 2 != 0 and n % 3 != 0)
        particion = true;

    return particion;
}
```

Ejemplo 48. Si obtienes todos los números que es posible crear multiplicando únicamente los primos 2, 3 y 5, y tomas aquéllos menores que un entero positivo dado L, donde $L \leq 10^{10}$, ¿cuántos de esos números cumplen con la condición de que la suma de sus divisores, a excepción de sí mismo, es mayor que él?

Solución. ¿Has escuchado hablar de los números perfectos? Son enteros positivos que igualan la suma de sus divisores, excluyéndose a sí mismo. Si bien es cierto que queremos que la suma de los divisores supere al número, podemos comenzar con este concepto.

Si investigas un poco, verás que los únicos números perfectos menores que 10^{10} son 6, 28, 496, 8128 y 33550336, y sus factorizaciones se pueden escribir como sigue:

$$6 = 2 \times 3$$
$$28 = 2^2 \times 7$$
$$496 = 2^4 \times 31$$

$$8128 = 2^6 \times 127$$
$$33550336 = 2^{12} \times 8191$$

Nota que el único número perfecto que incluyen únicamente primos permitidos (2, 3, 5) es 6, por lo que el resto no nos sirven.

Es claro que $6 = 1 + 2 + 3$. Ahora bien, cualquier natural mayor que 6 que sea múltiplo de 6 se puede escribir como $6x$, para $x \in \mathbb{N}, x \geq 2$, y $6x = x + 2x + 3x$; obviamente $x \mid 6x$, $2x \mid 6x$ y $3x \mid 6x$. No obstante, en esa suma no estamos contemplando todos los divisores de $6x$; independientemente de cuál sea el valor de x, 1 siempre dividirá a $6x$, y es distinto a x, $2x$ y $3x$. Aclaremos que, aunque puede que no sea sólo 1 el divisor faltante en la suma, sí nos ayuda a asegurar que $6x <$ *suma de divisores*. Así, todos los múltiplos de 6 distintos de 6 cumplen con la condición requerida.

Para saber cuántos enteros son, pensemos en que, dentro de los naturales, cada 6 números hay un múltiplo de 6, por lo que hay $L/6$ enteros positivos divisibles por 6 menores o iguales que L; por otro lado, nota que $L/6$ podría no ser entero, lo que implica que el siguiente múltiplo de 6 no fue "alcanzado", y deberemos quedarnos con la parte entera de $L/6$, que se puede obtener con la función *piso*. Finalmente, recordemos que 6 debe ser omitido del conteo, pues la suma de sus divisores lo iguala, no lo supera. Así, la fórmula que buscamos es $\lfloor L/6 \rfloor - 1$.

```
long long solucion()
{
    long long L;
    cin>>L;
    return floor(L/6) - 1;
}
```

Ejemplo 49. ¿Cuál es el resultado de la división $\frac{a^n - b^n}{a-b}$, para $n \in \mathbb{N}$ y $a, b \in \mathbb{Z}$? Si el resultado no es entero, regresa 0.

Solución. Ya vimos que siempre se cumple que $a - b \mid a^n - b^n$, más aún:

$$a^n - b^n = (a - b)(a^{n-1} + a^{n-2}b + \cdots + ab^{n-2} + b^{n-1})$$

De aquí,

$$\frac{a^n - b^n}{a - b} = (a^{n-1} + a^{n-2}b + \cdots + ab^{n-2} + b^{n-1})$$

```
int solucion()
{
    int a, b, n, i, respuesta = 0;
    cin>>a>>b>>n;

    for (i = 1; i <= n; i = i + 1)
        respuesta = respuesta + potencia(a, n-i) * potencia(b, i-1);

    return respuesta;
}
```

Ejemplo 50. ¿Cuál es el resultado de la división $\frac{a^n+b^n}{a+b}$, para $n \in \mathbb{N}$ y $a, b \in \mathbb{Z}$? Si el resultado no es entero, regresa 0.

Solución. Según lo estudiado, sabemos que si n es par, no hay una solución entera, y ése es el caso en el que regresaremos 0.

Por otro lado, cuando n es impar,

$$a^n+b^n = a^n-(-b)^n = (a-(-b))(a^{n-1}+a^{n-1}(-b)+\cdots+a(-b)^{n-2}+(-b)^{n-1})$$

Lo que nos conduce a:

$$\frac{a^n + b^n}{a + b} = a^{n-1} + a^{n-1}(-b) + \cdots + a(-b)^{n-2} + (-b)^{n-1}$$

```
void solucion()
{
    int a, b, n, i, respuesta = 0;
    cin>>a>>b>>n;

    if (n % 2 == 0)
        return respuesta;

    for (i = 1; i <= n; i = i + 1)
        respuesta = respuesta + potencia(a, n-i) * potencia(-b, i-1);

    return respuesta;
}
```

Es tremendamente usual enfrentarse a problemas en los que se requiera encontrar todos los divisores de un entero. La idea natural para obteneros podría ser recorrer todos los enteros menores o iguales que n y verificar, para cada uno, si éste es un divisor de n, lo que resultaría en una complejidad de $O(n)$. No obstante, la proposición enunciada a continuación nos proporcionará una manera de reducir la complejidad a $O(\sqrt{n})$.

Proposición 8. Sea n un número natural. Cada producto de exactamente dos enteros positivos distintos que iguala a n se compone de un natural menor que $\sqrt{n}$ y uno mayor que $\sqrt{n}$.

Demostración. Sea $a_1 \cdot a_2$ una factorización en dos elementos distintos de n. De inicio, existen tres posibilidades:

1. Que $a_1 < \sqrt{n}$ y $a_2 < \sqrt{n}$.
2. Que $a_1 > \sqrt{n}$ y $a_1 > \sqrt{n}$.
3. Que $a_1 < \sqrt{n}$ y $a_2 > \sqrt{n}$.

En el primer caso, si multiplicamos a_1 y a_2, tendremos:

$$a_1 \cdot a_2 < \sqrt{n}\sqrt{n} \Longrightarrow a_1 \cdot a_2 < n$$

En el segundo caso:

$$a_1 \cdot a_2 > \sqrt{n} \cdot \sqrt{n} \Longrightarrow a_1 \cdot a_2 > n$$

Ninguna de las dos primeras opciones permite obtener el valor de n, con lo que las descartamos, con lo que únicamente resta la opción 3, en la que uno de los factores es menor que $\sqrt{n}$ y el otro es mayor.

□

Es fácil ver que el único caso en el que se tiene un producto de dos enteros iguales que resulten en n, ocurre cuando n es un cuadrado perfecto, y los mencionados divisores serán $\sqrt{n}$.

De esta manera, para encontrar los factores de n, bastará con recorrer los enteros hasta $\sqrt{n}$, inclusive, y si se cumple que $i \mid n$, consideraremos a i y a n/i divisores de n.

Ejemplo 51. Obtén la suma de todos los divisores de un número natural dado n.

Solución. Realizaremos la búsqueda hasta la parte entera de $\sqrt{n}$, es decir, hasta la función piso. Hay que aclarar que nosotros no estamos invocando a dicha función porque guardamos la raíz en un entero, tomando así la parte entera en automático.

```
vector<int> divisores(int n)
{
    int i, raiz = sqrt(n);
    vector<int> resp;

    for (i = 0; i <= raiz; i = i + 1)
        if (n % i == 0)
        {
            resp.push_back(i);

            //Si n no es cuadrado perfecto:
            if (i != n/i)
                resp.push_back(n/i);
        }

    return resp;
}

int solucion()
{
    int n, suma = 0;
    cin>>n;
    auto divisores = divisores(n);

    for (int d: divisores)
        suma = suma + d;

    return suma;
}
```

2.2. Primos

Factorizar un entero n significa encontrar dos o más divisores de él cuya multiplicación sea igual a n. En particular, es de interés la *factorización por primos*, pues si se obtienen todos los factores primos de n, es posible obtener los restantes; además, la factorización por primos es la descomposición con más divisores que se puede obtener de n.

Existe una definición muy conocida de un número primo: Un número que únicamente es divisible entre 1 y sí mismo. No obstante, esta definición nos conduce a otra pregunta: ¿1 es primo? La respuesta es no, y para evitar confusiones, definiremos a un número primo como *un entero que es divisible por exactamente dos números naturales distintos.*

Probemos primero que todo número natural tiene una factorización por primos única.

Teorema 8. [Teorema Fundamental de la Aritmética]. Sea n un número natural mayor que 1. Existe una factorización única, salvo el orden, para

$$n = p_1 \cdot p_2 \cdot p_3 \cdot \cdots \cdot p_k$$

Donde p_1, p_2, p_3, $\ldots$, p_k son todos números primos no necesariamente distintos.

Demostración. Si n es primo, el teorema está demostrado, así que lo que hay que verificar es que se cumpla cuando n es un número compuesto.

Ya que n no es primo, es posible expresarlo como el producto de dos enteros distintos de 1 y sí mismo. Podemos asegurar que uno de ellos es el menor de los divisores de n (no necesariamente único, y excluyendo a 1), y que además es primo (pues si no lo fuera, podría escribirse como el producto de dos números menores que él, que, a su vez, serían divisores que n, lo que sería una contradicción), así que $n = p_1 \cdot n_1$, para algún natural $n_1 \neq 1$; ahora, si n_1 es primo, el teorema está demostrado, si no, podemos escribir a n_1 como $p_2 \cdot n_2$, para algún natural $n_2 \neq 1$ y p_2 el segundo divisor más pequeño de n. Si n_2 es un número primo, terminamos, de lo contrario, descomponemos de la misma forma a n_2, n_3, n_4, $\ldots$, y considerando que el conjunto de números primos menores que n es finito, porque la cantidad de naturales menores que n lo es, obtendremos una factorización de n como sigue:

$$n = p_1 \cdot p_2 \cdots p_k \cdot n_k$$

Donde o $n_k = 1$, o bien, n_k es un primo.

Por tanto, existen dos posibilidades: $n = p_1 \cdot p_2 \cdots p_k \cdot p_{k+1}$ (si n_k es primo) o $n = p_1 \cdot p_2 \cdots p_k$ (si $n_k = 1$).

Luego, para probar la unicidad de la factorización, supongamos que existen dos descomposiciones por primos distintas para n: $p_1 \cdot p_2 \cdots p_k$ y $q_2 \cdot q_3 \cdots q_r$; recordemos que establecimos a p_1 como el menor de los divisores de n distintos de 1. Se sigue que:

$$\begin{aligned} n = p_1(p_2 \cdots p_k) &\Longrightarrow p_1 \Big| n \\ &\Longrightarrow p_1 \Big| q_1 \cdot q_2 \cdots q_r \\ &\Longrightarrow p_1 \Big| q_j \text{ en donde } q_j \text{ es alguno de los factores } q_1, q_2, \ldots, q_r. \\ &\Longrightarrow p_1 = q_j \text{ porque } q_j \text{ es primo y distinto de 1.} \end{aligned}$$

Por comodidad, asumiremos que $j = 1$. Entonces,

$$\begin{aligned} & p_2 \cdot p_3 \cdots p_k = q_2 \cdot q_3 \cdots q_r \\ \Longrightarrow & p_2 \Big| q_j \text{ en donde } q_j \text{ es alguno de los factores } q_1, q_2, \ldots, q_r. \\ \Longrightarrow & p_2 = q_j \text{ porque } q_j \text{ es primo y distinto de 1.} \end{aligned}$$

Luego, asumiremos que $j = 2$. Y $p_3 \cdot p_4 \cdots p_k = q_3 \cdot q_4 \cdots q_r$.

Si repetimos el proceso $k - 1$ veces y suponiendo, sin pérdida de generalidad, que $k \leq r$, obtendremos que $p_k = q_k \cdot q_{k+1} \cdots q_r$.

Siguiendo el razonamiento anterior, $p_k \Big| q_k$, entonces $q_{k+1} \cdots q_r = 1$, y para que esto pase, $q_{k+1} = q_{k+2} = \cdots = q_r = 1$, lo que significa que $k = r$.

□

Luego, agrupando todos los primos iguales, llegaremos al siguiente resultado.

Para un número natural n, existe una factorización por primos única, salvo el orden:

$$n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \ldots \cdot p_k^{\alpha_k}$$

Donde todos los p_i son primos distintos entre sí y $\alpha_i \in \mathbb{N}$, para $i = 1, 2, \ldots, k$.

Proposición 9. Existe una cantidad infinita de números primos.

Demostración. [Prueba de Euclides]. Supongamos que existe un conjunto finito $P = \{p_1, p_2, \ldots, p_k\}$ que contiene a todos los primos. Entonces debe existir un entero positivo que resulte del producto de todos los elementos en P, mismo al que nombraremos M. Luego, $M+1$ puede ser primo o no. Si lo es, será un primo que no está dentro de P, lo que sería una contradicción, así que analicemos la posibilidad de que no lo sea.

Ya que estamos suponiendo que $M + 1$ no es primo, debe existir algún $p_i \in P$ que divida a $M+1$, pero ya habíamos establecido que M era múltiplo de p_i. Se sigue entonces que $p_i \Big| (M+1) - M \implies p_i \Big| 1$, el problema es que no existe un primo del que 1 sea múltiplo, llegando así a otra contradicción. Concluimos entonces que no existe un conjunto finito que contenga a todos los primos.

□

Cuando queremos obtener "a mano" los factores primos de un número, utilizamos una técnica aparentemente muy sencilla. Factoricemos, por ejemplo, el número 60:

$$\begin{array}{r|l} 60 & 2 \\ 30 & 2 \\ 15 & 3 \\ 5 & 5 \\ 1 & \end{array}$$

Lo que nos dice que $60 = 2^2 \cdot 3 \cdot 5$. Este producto al que llegamos es nombrado factorización por primos, y si está escrito como un producto de potencias de primos ordenados de menor a mayor, recibe también el nombre de *descomposición canónica*.

Para una computadora, sin embargo, este procedimiento no resulta tan sencillo,

pues, de inicio, no tiene conocimiento de cuáles enteros son primos; comprobar que un número n es primo con el método más sencillo requiere verificar, para cada entero positivo distinto de 1 menor o igual que $\sqrt{n}$ que $n \,\%\, i \neq 0$ (por lo demostrado en la proposición 8), y la complejidad de este procedimiento es $O(\sqrt{n})$. Por tanto, en un algoritmo para hallar los divisores primos de un natural n, lo primero que se requiere es encontrar todos los primos menores o iguales a n, para después seleccionar aquéllos de los que n es múltiplo.

La siguiente técnica nos permite, dado un entero positivo n, hallar a todos los primos menores o iguales que éste de una manera más eficiente que la idea expuesta en el párrafo anterior.

2.2.1. Criba de Eratóstenes

El algoritmo de la Criba de Eratóstenes comienza con una lista de todos los naturales desde 2 hasta n, sobre la que iteraremos, y en cada iteración, encontraremos un primo mayor al anterior. En la primera pasada, sabemos que 2 es primo, pero sus múltiplos distintos a él no lo son, por lo que serán tachados; en la segunda pasada, el siguiente entero disponible es 3, con lo que sabemos que 3 es primo, pero no sus múltiplos mayores a 3, así que se tacharán también; el siguiente primo será 5 (que es el siguiente elemento sin eliminar), y también se tacharán los múltiplos de 5 mayores a él. Entonces, generalizando, en cada pasada, tomaremos como primo al siguiente número *no tachado* y sacaremos del juego a todos sus múltiplos distintos a él; esto es equivalente a tachar un número cada p enteros, donde p es el primo hallado.

Encontremos, por ejemplo, todos los números primos menores o iguales a 12 haciendo uso de la criba.

Lista:	$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12$
Primera pasada:	$\mathbf{2}, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}, 11, \cancel{12}$
Segunda pasada:	$\mathbf{2}, \mathbf{3}, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}$

Tercera pasada:	$\mathbf{2}, \mathbf{3}, \cancel{4}, \mathbf{5}, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}$
Cuarta pasada:	$\mathbf{2}, \mathbf{3}, \cancel{4}, \mathbf{5}, \cancel{6}, \mathbf{7}, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}$
Quinta pasada:	$\mathbf{2}, \mathbf{3}, \cancel{4}, \mathbf{5}, \cancel{6}, \mathbf{7}, \cancel{8}, \cancel{9}, \cancel{10}, \mathbf{11}, \cancel{12}$

Y los primos son los números que quedaron sin tachar: 2, 3, 5, 7 y 11.

Nota que varios números fueron tachados más de una vez, lo que nos da una pauta para decir que el algoritmo aún puede ser optimizado: Cuando eliminamos los múltiplos de un primo p, eliminamos los números de la forma $p \cdot 2$, $p \cdot 3$, ..., pero si el otro factor es menor que p, quiere decir, que en alguna ronda anterior, éste ya fue tachado. De ahí sabemos que, en cada ronda, podemos comenzar a tachar en el número $p \cdot p$.

Ejemplo 52. Encuentra todos los primos menores o iguales que un entero positivo n dado.

Solución. Para la implementación de la criba de Eratóstenes recurriremos al uso de un *bitset*, en el que la posición i tomará valor de 1 si i es primo, y de 0 en caso contrario.

```
1  vector<int> primos;
2  bitset<10000000> es_primo;
3
4  void criba()
5  {
6      long long i, j, n;
7      es_primo.set(); //Da el valor de 1 a todo el bitset.
8      es_primo[0] = es_primo[1] = 0; //0 y 1 no son primos.
9      cin>>n;
10
11     for (i = 2; i <= n; i = i + 1)
12         if (es_primo[i])
13         {
14             primos.push_back(i);
15
16             //Los múltiplos de i mayores a i "se tachan".
17             for (j = i * i; j <= n; j = j + i)
18                 es_primo[j] = 0;
19         }
20 }
```

Dado que los números primos siempre serán los mismos, la criba puede ser ejecutada como un preproceso, obteniendo los primos menores o iguales que el máximo valor que puedes recibir como entrada. De esta forma, para la obtención de un primo, únicamente será necesaria una consulta, misma que se realizará en tiempo constante.

Ejemplo 53. Determina los números primos que dividen a un entero positivo n.

Solución. En primer lugar, hay que usar la Criba de Eratóstenes para encontrar los primos menores o iguales que n, para lo que únicamente invocaremos el vector de primos obtenido en la función del ejemplo 52 (nota que fue declarado global). Luego, verificaremos si n es divisible por cada elemento de dicho vector.

```
1  vector<int> divisores_primos()
2  {
3      long long n;
4      vector<int> primos_n;
5      cin>>n;
6
7      for (int p: primos)
8      {
9          if (n % p == 0)
10             primos_n.push_back(p);
11
12         if (p * p > n)
13             break;
14     }
15
16     return primos_n;
17 }
```

Ejemplo 54. Obtén la descomposición canónica de un número natural n.

Solución. En este ejercicio, nuevamente vamos a requerir la criba; para cada primo comprobaremos si es divisor de n, y si lo es, realizaremos la operación $n = n/primo$ hasta que n resulte en un número no divisible por el primo en cuestión, en cuyo caso continuaremos con el primo siguiente. Realizaremos este proceso hasta que $n = 1$.

```
vector<pair<int, int>> desc_canonica()
{
    long long n;
    int exponente;
    vector<pair<int, int>> factores_primos;
    cin>>n;

    for (int p: primos)
    {
        if (n % p == 0)
        {
            exponente = 0;
            while (n % p == 0)
            {
                n = n / p;
                exponente = exponente + 1;
            }
        }

        factores_primos.push_back({p, exponente});

        if (p * p > n)
            break;
    }

    return factores_primos;
}
```

Si deseamos imprimir la descomposición canónica, en lugar de regresar el vector de pares, agregaremos las siguientes líneas al programa:

```
void desc_canonica()
{
    int i;
    ...

    cout<<factores_primos[0].first<<"^"<<fact_primos[0].second;
    for (i = 1, i < factores_primos.size(); i = i + 1)
        cout<<" * "<<factores_primos[i].first
            <<"^"<<factores_primos[i].second;
}
```

Ejemplo 55. Dados tres enteros positivos m, n y un primo p, ¿para cuántos valores naturales de k mayores que 1 se cumple la siguiente expresión? (Problema modificado de "Es lo mismo pero diferente", tomado de [3]).

$$p^k \left| \sum_{i=0}^{n} \binom{n}{i} m^i \right.$$

En caso de no existir ningún valor válido para k, devuelve -1.

Solución. La suma que se muestra en la expresión es igual a $(m+1)^n$ (ver la sección de *Coeficientes Binomiales* en el capítulo de *Análisis Combinatorio* si esto no es claro), por tanto, buscamos valores de k para los que $p^k \left| (m+1)^n \right.$.

Para que un primo divida a una potencia de $m+1$, es necesario que divida a $m+1$, lo que nos lleva a deducir que lo primero que debemos verificar es que $p \left| m+1 \right.$. Si esto no ocurre, no hay k que satisfaga lo pedido e imprimiremos -1, de lo contrario, continuaremos.

Luego, habrá que ver cuántas veces podemos dividir a $m+1$ entre p, nombremos a este número t, y sabremos que $m+1 = p^t \cdot x$, para algún entero x. Así pues, estamos buscando valores de k tales que $p^k \left| p^t \cdot x \right.$; ya que t incluye todas las veces en que p es factor de $m+1$, podemos estar seguros de que $p^k \not| x$, por lo que la única opción es que $p^k \left| p^t \right.$, algo que solamente ocurrirá cuando $k \leq t$. Se concluye entonces que, todos los naturales entre 2 y t, inclusive, cumplen la condición, lo que implica que la respuesta será $t-1$.

```
1  int solucion()
2  {
3      int m, n, p, S, t = 0;
4      cin>>m>>n>>p;
5      S = m + 1;
6
7      if (S % p != 0)
8          return -1;
9
10     while (S % p == 0)
11     {
```

```
12            S = S / p;
13            t = t + 1;
14        }
15
16        return t - 1;
17    }
```

La complejidad del algoritmo de la criba de Eratóstenes para algún natural n es $O(n \cdot log(log\ n))$: Para cada primo encontrado, se tachan n/p números, así, para k primos, se realiza el siguiente número de operaciones:

$$\frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{p_k} = n\left[\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{p_k}\right]$$

Reducir la expresión a la que llegamos queda fuera del alcance de esta obra, por lo que únicamente haremos uso del hecho de que $1/2 + 1/3 + \cdots + 1/p_k \approx log(log\ n)$, mencionado en [11]. Así, la criba puede ser ejecutada dentro de tiempo hasta 10^7.

Si ya contamos con la factorización por primos de un entero n, podemos obtener todos los divisores compuestos de n haciendo todas las multiplicaciones posibles de los factores primos. Por ejemplo, la descomposición canónica de 18 es $2 \cdot 3^2$, lo que nos dice que sus divisores compuestos son $2 \cdot 3$, $3 \cdot 3$ y $2 \cdot 3 \cdot 3$. En total, 18 tiene 6 factores distintos, incluyendo a 1 y a sí mismo. Entonces, si requerimos conocer la cantidad de divisores de un número, ¿es necesario realizar todas las multiplicaciones para después contarlas? La siguiente proposición nos dará la respuesta a esa pregunta.

Proposición 10. Si $p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdots p_k^{\alpha_k}$ es la descomposición canónica de algún entero $n > 1$, el número de divisores positivos de n es:

$$(\alpha_1 + 1)(\alpha_2 + 1) \cdots (\alpha_k + 1)$$

Demostración. Cada divisor compuesto de n es el producto de una subsecuencia de los primos que lo dividen, así que para obtener el total de factores habrá que contar el total de subsecuencias de primos que es posible crear.

Cada potencia $p_i^{\alpha_i}$ implica que disponemos de α_i primos p_i, lo que nos lleva a que, para crear una lista de factores primos, tenemos $\alpha_i + 1$ opciones con respecto a p_i: tomarlo 0 veces, 1 vez, 2 veces, ..., α_i veces. Luego, por el principio fundamental de conteo y considerando que se tienen k primos distintos, el total de divisores de n es $\prod_{i=1}^{k} (\alpha_i + 1)$.

□

Ejemplo 56. Dados dos números naturales m y n, determina cuál de los dos tiene más divisores distintos; si tienen la misma cantidad de factores, imprime 0.

> **Solución.** Lo primero que hay que hacer es obtener la descomposición canónica de m y de n para luego aplicar la fórmula que obtuvimos en la proposición 10. También haremos uso del vector de primos obtenido en la función de la criba que ya antes codificamos.

```
1  vector<int> exponentes_primos(int n)
2  {
3      int exp, i;
4      vector<int> exponentes;
5
6      for (int p: primos)
7      {
8          if (n % p == 0)
9          {
10             exp = 0;
11             while (n % p == 0)
12             {
13                 n = n / p;
14                 exp = exp + 1;
15             }
16         }
17         exponentes.push_back(exp);
18
19         if (p * p > n)
20             break;
21     }
22
23     return exponentes;
24 }
25
26 int solucion()
```

```
{
    int m, n, prod_m = 1, prod_n = 1;
    cin>>m>>n;
    auto exp_m = exponentes_primos(m);
    auto exp_n = exponentes_primos(n);

    for (int exp: exp_m)
        prod_m = prod_m * (exp + 1);

    for (int exp: exp_n)
        prod_n = prod_n * (exp + 1);

    if (prod_m > prod_n)
        return m;
    else if (prod_m < prod_n)
        return n;
    else
        return 0;
}
```

Ejemplo 57. Dado un número natural n, ¿cuántos divisores positivos tiene $n!$?

Solución. En este problema ejecutaremos la criba para obtener los primos menores o iguales que n. ¿Por qué no de $n!$? $n! = n(n-1)\cdots(2)(1)$, por tanto, todos los primos que dividan a $n!$ deben dividir a alguno de los naturales menores o iguales que n, lo que implica que ningún primo que divida a $n!$ será mayor a n.

Ahora procederemos a buscar la descomposición en primos de $n!$: Tomemos, por ejemplo, el caso del 2. Cada exactamente 2 enteros hay un múltiplo de 2, por lo que en $\{1, 2, 3, \ldots, n\}$ existen $\lfloor n/2 \rfloor$ números pares, lo que implica que $n!$ tiene como factor a $2^{\lfloor n/2 \rfloor}$; no obstante, no hemos contado todos los factores 2 que se encuentran en la descomposición canónica de $n!$, pues existen múltiplos de 2 que, además, son múltiplos de 4 (2^2), de 8 (2^3), de 16 (2^4), etcétera. Siguiendo la misma lógica que para 2, existen $\lfloor n/4 \rfloor$ múltiplos de 4 en los primeros n naturales, $\lfloor n/8 \rfloor$ múltiplos de 8 y así sucesivamente. Así, para considerar todos los factores 2 que conforman a $n!$, hay que considerar todos los sumandos: $\lfloor n/2 \rfloor + \lfloor n/2^2 \rfloor + \lfloor n/2^3 \rfloor + \cdots + \lfloor n/2^{k_2} \rfloor$, en donde k_2 es el último valor

para que el que $n/2^{k_2}$ es mayor o igual que 1. Llamaremos F_2 al exponente que corresponde a 2 en la descomposición canónica de $n!$.
El procedimiento que aplicamos para 2 deberá llevarse a cabo para todos los primos menores o iguales que n; en general:

$$F_p = \sum_{i=1}^{k_p} \left\lfloor \frac{n}{p^i} \right\rfloor$$

En donde k_p es el último número natural para el que n/p^{k_p} es mayor o igual a 1.

Finalmente, el total de divisores de $n!$ será:

$$(F_2 + 1)(F_3 + 1)(F_5 + 1) \cdots (F_q + 1)$$

En donde q es el mayor primo menor o igual que n.

```
vector<int> obtener_F(int n)
{
    int F;
    vector<int> potencias;

    for (int p : primos)
    {
        if (p > n)
            break;

        F = 0;
        p_i = p;
        while (n / p_i >= 1)
        {
            F = F + n / p_i;
            p_i = p_i * p;
        }

        potencias.push_back(F);
    }

    return F;
}

int solucion()
```

```
26  {
27      int n, respuesta = 1;
28      cin>>n;
29      obtener_F(n);
30
31      for (int pot : potencias)
32          respuesta = respuesta * (pot + 1)
33
34      return respuesta;
35  }
```

Proposición 11. Si $p_1^{\alpha_1} \cdot p_2^{\alpha_1} \cdot \cdots p_k^{\alpha_k}$ es la factorización por primos de un número natural n, la suma de los divisores positivos de n es:

$$\prod_{i=0}^{k} \frac{p_i^{\alpha_i+1} - 1}{p_i - 1}$$

Demostración. Pensemos primero en la suma de los divisores de un número de la forma p^α, para p primo. Sus únicos factores son p^0, p^1, ..., p^α; sumar estos números corresponde a la suma de una progresión geométrica en la que la razón es p, lo que nos lleva a deducir que la suma de los divisores de p^α es $(p^{\alpha+1} - 1)/(p - 1)$, a la que denotaremos como $\sigma(p^\alpha)$.

Si probamos que la función $\sigma(p^\alpha)$ es una *función multiplicativa*[1], habremos terminado, pues podremos aplicar esta propiedad a la factorización por primos de n.

Consideremos el producto $p_i^{\alpha_i} \cdot p_j^{\alpha_j}$ $(1 \leq i \neq j \leq k)$, que, al estar compuesto por únicamente primos, es seguro que $p_i^{\alpha_i}$ y $p_j^{\alpha_j}$ no comparten divisores, salvo 1, y que cualquier número que divida a dicho producto estará compuesto por una potencia de p multiplicada por una potencia de q. Luego, podemos escribir la suma de todos sus divisores como se muestra a continuación.

$$\sigma(p_i^{\alpha_i} \cdot p_j^{\alpha_j}) = p_i^0 p_j^0 \ + \ p_i^1 p_j^0 \ + \ p_i^2 p_j^0 \ + \cdots + \ p_i^{\alpha_i} q_j^0$$

$$+ \, p_i^0 q_j^1 \ + \ p_i^1 q_j^1 \ + \ p_i^2 q_j^1 \ + \cdots + \ p_i^{\alpha_i} q_j^1$$

[1] Sean a y b dos números que no tienen factores en común, salvo 1. Decimos que una función f es multiplicativa si $f(a \cdot b) = f(a) \cdot f(b)$

$$
\begin{aligned}
&+ p_i^0 q_j^2 \ + \ p_i^1 q_j^2 \ + \ p_i^2 q_j^2 \ + \cdots + \ p_i^{\alpha_i} q_j^2 \\
&\vdots \\
&+ p_i^0 q_j^{\alpha_j} + p_i^1 q_j^{\alpha_j} + p_i^2 q_j^{\alpha_j} + \cdots + p_i^{\alpha_i} q_j^{\alpha_j} \\
&= p_j^0 \sum_{k=0}^{\alpha_i} p_i^k + p_j \sum_{k=0}^{\alpha_i} p_i^k + p_j^2 \sum_{k=0}^{\alpha_i} p_i^k + \cdots + p_j^{\alpha_j} \sum_{k=0}^{\alpha_i} p_i^k \\
&= \sum_{k=0}^{\alpha_i} p_i^k \cdot \left[p_j^0 + p_j^1 + p_j^2 + \cdots + p_j^{\alpha_j} \right] \\
&= \sum_{k=0}^{\alpha^i} p_i^k \cdot \sum_{l=0}^{\alpha^j} p_j^l = \sigma(p_i^{\alpha_i}) \cdot \sigma(p_j^{\alpha_j})
\end{aligned}
$$

Con lo anterior, hemos probado que la suma de los divisores de un número es una función multiplicativa. Por tanto, y ya que los factores primos no tienen divisores en común además de 1, se cumplirá lo siguiente:

$$
\begin{aligned}
&\sigma(p_1^{\alpha_1} \cdot p_2^{\alpha_1} \cdots p_k^{\alpha_k}) = \sigma(p_1^{\alpha_1}) \cdot \sigma(p_2^{\alpha_1}) \cdot \cdots \cdot \sigma(p_k^{\alpha_k}) \\
&\implies \sigma(n) = \left(\frac{p_1^{\alpha_1+1} - 1}{p_1 - 1} \right) \left(\frac{p_2^{\alpha_2+1} - 1}{p_2 - 1} \right) \cdots \left(\frac{p_k^{\alpha_k+1} - 1}{p_k - 1} \right)
\end{aligned}
$$

□

Ejemplo 58. Dado un número natural n mayor que 1, determina si éste es deficiente, perfecto o abundante.

Un número es deficiente si la suma de sus divisores propios[2] es menor que n, es perfecto si la suma de sus divisores propios es n y es abundante si la suma de sus divisores propios es mayor que n.

Solución. Lo primero será obtener la descomposición canónica de n, seguido de aplicar la fórmula a la que llegamos en la proposición 11, no

[2] Los divisores propios de un número natural n son aquéllos menores que n.

obstante, ésta requerirá una pequeña modificación para excluir al divisor impropio de n (que es n mismo). Así, la suma de los divisores propios de n será $\sigma(n) - n$; obtenida esa expresión, lo único que resta será comparar el resultado con n para categorizarlo como deficiente, perfecto o abundante.

```
void solucion()
{
    int n, sigma_n = 1;
    cin>>n;
    desc_canonica(n);

    for (auto par : factores_primos)
        sigma_n = sigma_n * ( potencia(par.first, par.second + 1) - 1 )
                / (par.first - 1);

    if (simga_n - n < n)
        cout<<"Deficiente.\n";
    else if (sigma_n - n = n)
        cout<<"Perfecto.\n";
    else
        cout<<"Abundante.\n";
}
```

Ejemplo 59. Dado un número natural n mayor que 1, obtén la suma de sus divisores pares.

Solución. Para que n tenga factores pares, n debe ser par; si no lo es, nuestra respuesta será 0.

En caso de que 2 sí divida a n, cuando obtengamos su descomposición canónica, $p_1^{\alpha_1} = 2^{\alpha_1}$. Recordemos que, cuando hicimos la demostración de la proposición 11, desglosamos todas las sumas de un producto. Pues bien, para que un divisor sea par, es necesario que la potencia de 2 sea mayor o igual que 1, por lo que omitiremos todos los sumandos en los que se encuentre 2^0, modificándose únicamente $\sigma(2^{\alpha_1})$:

$$\sigma'(2^{\alpha_1}) = 2^1 + 2^2 + \cdots + 2^{\alpha_1}$$

$$= 2^{\alpha_1+1} - 1 - 2^0 = 2^{\alpha_1+1} - 2$$

Finalmente, la suma de los divisores pares resulta en:

$$\sigma_{pares}(n) = \left(2^{\alpha_1+1} - 2\right)\left(\frac{p_2^{\alpha_2+1} - 1}{p_2 - 1}\right) \cdots \left(\frac{p_k^{\alpha_k+1} - 1}{p_k - 1}\right)$$

```
int solucion()
{
    int i, n, s_par;
    cin>>n;

    if (n % 2 != 0)
        return 0;

    factores = desc_canonica(n);

    s_par = potencia(2, factores[0].second + 1) - 2;
    for (i = 1; i < fact.size(); i = i + 1)
        s_par = s_par * (potencia(factores[i].first,
                factores[i].second + 1) - 1) / (factores[i].first - 1);

    return s_par;
}
```

2.3. Aritmética modular

Aunque hay otras maneras de comprobarlo, nota que, en los ejemplos abordados en la sección de *Divisibilidad* y algunos de *Primos*, para saber si un número es múltiplo de otro, se usa la operación %, nombrada *módulo*.

Es probable que ya hayas tenido algún acercamiento con esta operación en el ámbito de la programación, sin embargo, debes saber que hay toda una aritmética que se desarrolla alrededor de ella, y es precisamente lo que estudiaremos en esta sección.

La aritmética modular tiene numerosas aplicaciones, como lo es la obtención de números pseudoaleatorios, el procesamiento de información con grandes números y

la criptografía, mismos que abordaremos más adelante.

Se dice que dos números enteros a y b son congruentes módulo n si al dividir ambos números entre n arrojan el mismo residuo, para $n \in \mathbb{N}$. En matemáticas, la notación utilizada es $a \equiv b \ (mod\ n)$, y se lee como "a es congruente con b módulo n". Es claro que hay una infinidad de números que cumplen con la condición de tener el mismo residuo módulo n, y todos ellos pertenecen a la misma clase de equivalencia (más adelante, probaremos que la operación módulo da lugar a una relación de equivalencia). Esto implica que cualquier número perteneciente a dicha clase puede ser colocado tanto de lado izquierdo como de lado derecho del signo $\equiv$, aunque claro, suele ser de utilidad trabajar con números lo más pequeños posible, por lo que es usual buscar el menor de los naturales en dicha clase de equivalencia.

En programación, por otro lado, trabajamos con la expresión $b = a \ \% \ n$, asignando a la variable b un único posible valor, que es el residuo de dividir a a entre n, es decir, el menor número dentro de la clase de equivalencia (del mismo signo que n). O sea, el valor de b será único y no podrá ser suplido por cualquier otro número perteneciente a su clase de equivalencia módulo n.

En esta sección abordaremos un enfoque matemático de los módulos, pero eso no significa que los resultados no puedan ser utilizados en programación, de hecho, los desarrollaremos justamente con el fin de que tengas dominio de ellos para aplicarlos en la resolución de los problemas que nos competen.

Volvamos entonces a la expresión $a \equiv b \ (mod\ n)$. Ya que sabemos que a y b tienen el mismo residuo módulo n, podemos escribirlos como $a = k_1 n + r$ y $b = k_2 n + r$, para $k_1, k_2 \in \mathbb{Z}$. Restándolos, $a - b = (k_1 n + r) - (k_2 n + r) = n(k_1 - k_2)$. De aquí, decir que $a \equiv b \ (mod\ n)$ es equivalente a afirmar que $n \big| a - b$.

Anteriormente dijimos que hay toda un área que se desarrolla alrededor de los módulos. Comencemos por probar que existe una relación de equivalencia sobre la operación módulo, probando su reflexividad, simetría y transitividad con el siguiente teorema.

Teorema 9. Sean a, b y c tres números enteros y sea n un número natural. Entonces:

1. Reflexividad: $a \equiv a \ (mod\ n)$.
2. Simetría: Si $a \equiv b \ (mod\ n)$, entonces $b \equiv a \ (mod\ n)$.
3. Transitividad: Si $a \equiv b \ (mod\ n)$ y $b \equiv c \ (mod\ n)$, entonces $a \equiv c \ (mod\ n)$.

Demostración. Ya vimos que si $a \equiv b \ (mod\ n)$, $n \Big| a-b$. Aprovecharemos este hecho para proceder con las demostraciones.

1. $a - a = 0$ y $0 = n \cdot 0$, por lo tanto, $n \Big| 0 \implies n \Big| a - a$, de lo que se deduce que $a \equiv a \ (mod\ n)$.
2. De $a \equiv b \ (mod\ n)$, se tiene $n \Big| (a - b)$. Luego, $n \Big| (-1)(a - b)$ (por lo visto en la sección de *Divisibilidad*), por lo que $n \Big| (b-a)$, que es lo mismo que $b \equiv a \ (mod\ n)$.
3. De $a \equiv b \ (mod\ n)$ y $b \equiv c \ (mod\ n)$, sabemos que $n \Big| (a - b)$ y $n \Big| (b - c)$. Sumando, resulta que $n \Big| (a-b)+(b-c) \implies n \Big| (a-c)$, que equivale a decir que $a \equiv c \ (mod\ n)$.

Probamos así que la operación módulo cumple lo necesario para ser una relación de equivalencia.

□

Veamos ahora las operaciones que se pueden realizar involucrando módulos.

Teorema 10. Sean a, b, c y d números enteros, y sea n un número natural. Se cumple:

1. Si $a \equiv b \ (mod\ n)$ y $c \equiv d \ (mod\ n)$, entonces $a + c \equiv b + d \ (mod\ n)$.
2. Si $a \equiv b \ (mod\ n)$ y $c \equiv d \ (mod\ n)$, entonces $a - c \equiv b - d \ (mod\ n)$.
3. Si $a \equiv b \ (mod\ n)$ y $c \equiv d \ (mod\ n)$, entonces $ac \equiv bd \ (mod\ n)$.
4. Si $a \equiv b \ (mod\ n)$, entonces $am \equiv bm \ (mod\ n)$, para cualquier entero m.
5. Si $a \equiv b \ (mod\ n)$, entonces $a^m \equiv b^m \ (mod\ n)$, para cualquier entero no negativo m.

Demostración. Nuevamente usaremos el hecho de que $a \equiv b \ (mod\ n)$ es equivalente a decir que $n \Big| (a - b)$.

1. $a \equiv b \ (mod\ n)$ y $c \equiv d \ (mod\ n)$ implica que $n\Big|(a-b)$ y $n\Big|(c-d)$. Sumando, obtenemos $n\Big|(a-b)+(c-d) \implies n\Big|(a+c)-(b+d)$, equivalente a $a+c \equiv b+d \ (mod\ n)$.

2. $a \equiv b \ (mod\ n)$ y $c \equiv d \ (mod\ n)$ implica que $n\Big|(a-b)$ y $n\Big|(c-d)$. Restando, obtenemos $n\Big|(a-b)-(c-d) \implies n\Big|(a-b-c+d) \implies n\Big|(a-c)-(b-d)$, equivalente a decir que $a-c \equiv b-d \ (mod\ n)$.

3. $a \equiv b \ (mod\ n)$ y $c \equiv d \ (mod\ n)$ implica que $n\Big|(a-b)$ y $n\Big|(c-d)$. Multiplicando $a-b$ por c, obtenemos $ac-bc$, y por las reglas de divisibilidad ya vistas, se deduce que $n\Big|(ac-bc)$. Análogamente, multiplicamos b por $c-d$, y también se llega a que $n\Big|(bc-bd)$. Por tanto, la suma de dichas expresiones es también múltiplo de n: $n\Big|(ac-bc)+(bc-bd) \implies n\Big|(ac-bd)$, es decir, $ac \equiv bd \ (mod\ n)$.

4. Se cumple que $n\Big|(a-b)$, entonces, $n|m(a-b) \implies n\Big|(am-bm)$, o sea, $am \equiv bm \ (mod\ n)$.

5. Directa del punto 3.

Nota que no hay una regla para involucrar divisiones. Más adelante abordaremos ese tema, pero por lo pronto, ten en cuenta que el hecho de que $a \equiv b \ (mod\ n)$ y que $c \equiv d \ (mod\ n)$ no tiene que implicar que $\frac{a}{c} \equiv \frac{b}{d} \ (mod\ n)$, o que $\frac{a}{m} \equiv \frac{b}{m} \ (mod\ n)$ para cualquier entero m.

□

El uso de módulos otorga diversas ventajas. Una de ellas es que los residuos son periódicos, lo que significa que su comportamiento es fácilmente analizable. Por otro lado, se gastan menos recursos al trabajar con los residuos que con el número original, pues un residuo módulo n siempre será menor (o igual), y únicamente puede tomar uno de los valores del conjunto $\{-(n-1), -(n-2), \ldots, -2, -1, 0, 1, 2, \ldots, n-2, n-1\}$; de esta forma, es posible trabajar con números sumamente grandes con tipos de datos relativamente pequeños.

Ejemplo 60. Obtén el entero no negativo más pequeño que es congruente con $38^{16} - 65(12) + 14$ módulo 7.

Solución. Para encontrar el resultado, dividiremos la expresión en partes:

$$[38^{16} - 65(12) + 14](mod\ 7) = 38^{16}(mod\ 7) - 65(12)(mod\ 7) + 14(mod\ 7)$$

Hecho esto, procedamos a obtener los residuos de cada parte.

Por la propiedad 5 de teorema 10, podemos realizar las siguientes operaciones:

$$\begin{aligned} 38 &\equiv 3\ (mod\ 7) \\ 38^2 \equiv 3^2 &\equiv 2\ (mod\ 7) \\ 38^4 \equiv 2^2 &\equiv 4\ (mod\ 7) \\ 38^8 \equiv 4^2 &\equiv 2\ (mod\ 7) \\ 38^{16} \equiv 2^2 &\equiv 4\ (mod\ 7) \end{aligned}$$

Por la propiedad 3 del teorema 10:

$$\begin{aligned} 65 \equiv 2\ (mod\ 7) \text{ y } 12 &\equiv 5\ (mod\ 7) \\ \implies 65(12) &\equiv 2(5) \equiv 3\ (mod\ 7) \end{aligned}$$

Finalmente:

$$14 \equiv 0\ (mod\ 7)$$

Por tanto:

$$[38^{16} - 65(12) + 14] \equiv 4 - 3 + 0\ (mod\ 7) = \mathbf{1}$$

Y así, convertimos operaciones cuyos resultados serían significativamente grandes al uso de enteros entre 0 y 6, inclusive, de una forma en que ni siquiera fue necesario calcular el valor real del resultado, pues era más sencillo encontrar su equivalente módulo 7.

Ejemplo 61. Obtén el residuo módulo 11 de la operación $11(41) + 40(2) - 14(7)$.

Solución. Particionando la expresión dada:

$$\begin{aligned}&[11(41) + 40(2) - 14(7)] \ (mod\ 11)\\ =&11(41) \ (mod\ 11) + 40(2) \ (mod\ 11) - 14(7) \ (mod\ 11)\end{aligned}$$

Ya que $11 \equiv 0 \ (mod\ 11)$, 11×41 también es congruente con 0.

Luego, $40 \equiv 7 \ (mod\ 11)$, por lo que $40(2) \equiv 7(2) \equiv 3 \ (mod\ 11)$.

Finalmente, $14 \equiv 3 \ (mod\ 11)$ y $7 \equiv 7 \ (mod\ 11)$. Entonces, $14(7) \equiv 3(7) \equiv 10 \ (mod\ 11)$.

Así, llegamos a que $[11(41) + 40(2) - 14(7)] \equiv 0 + 3 - 10 \ (mod\ 11) =$ **-7**.

El ejemplo anterior da lugar a dos interrogantes: ¿Por qué aplicar módulos a cada paso de una expresión si es posible aplicarlo únicamente al final?, ¿y qué sucede cuando te enfrentas con un resultado menor que 0 si requieres que la salida sea un entero no negativo?

Imagina que te encuentras con un problema en el que, en algún punto, tienes que resolver la operación 10^{24}, o incluso una mayor. El mayor entero que C es capaz de procesar cabe en un *long long* (entero de 64 bits) y es $9 \cdot 10^{18}$. Generalmente, cuando se requiere el uso de números mayores, se pide que se trabaje con su equivalencia en módulos, y aplicarlos a cada operación encontrada no solamente ayuda, sino que es indispensable para que el algoritmo funcione. Así, si estamos tratando con la expresión 10^{24}, al resolver 10^{19}, ya habremos superado el máximo valor posible y no podremos continuar con su resolución, sin embargo, si hacemos uso de la aritmética modular con algún módulo, digamos 7, y obtenemos los residuos a cada paso, tendremos un escenario como el que sigue.

$$\begin{aligned}10 &\equiv 3 \ (mod\ 7)\\ 10^2 &\equiv 3^2 \equiv 2 \ (mod\ 7)\\ 10^4 &\equiv 2^2 \equiv 4 \ (mod\ 7)\\ 10^8 &\equiv 4^2 \equiv 2 \ (mod\ 7)\end{aligned}$$

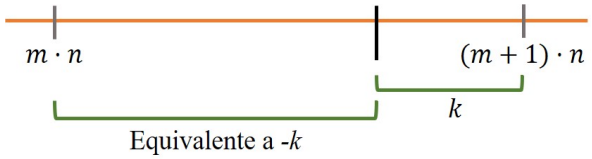
$$10^{16} \equiv 2^2 \equiv 4 \ (mod\ 7)$$

$$\implies 10^{24} = 10^{16} \cdot 10^8 \equiv 4(2) \equiv 1 \ (mod\ 7)$$

Como dijimos, no es posible guardar el 10^{24} en ningún entero en C, pero al aplicar módulos a cada paso de la operación, podemos obtener el equivalente modular de 10^{24} sin necesidad de guardar u obtener el resultado real.

En C++, la sentencia $b = a \ \% \ n$, para cualquier entero no negativo b arrojará un natural, o bien, 0 (si a es múltiplo de n), entonces, ¿cuándo es que se obtienen números menores que 0? En la práctica, es común encontrarlos cuando hay una resta involucrada, como en el caso del ejemplo 61; no obstante, otra cosa que es común en la práctica es que se trabaje únicamente con residuos no negativos, entonces la pregunta que sigue es ¿qué hacer si te enfrentas con una situación así?

Cada residuo negativo tiene una equivalencia dentro de los residuos positivos. Tomemos alguno de los residuos negativos módulo n, digamos $-k$, donde $0 \leq k \leq n-1$. La manera en que éste puede ser interpretado es que *faltan* k unidades para encontrar el siguiente múltiplo de n, por tanto, el equivalente del residuo $-k$ es el número de unidades que sobran desde el último múltiplo de n, digamos $m \times n$, para $m \in \mathbb{Z}$. Para obtener el equivalente positivo de un módulo negativo, se requiere únicamente sumar n después de aplicar el módulo; cuando sumemos n, obtendremos las unidades que *sobren* después del último múltiplo de n, esto es $(m \cdot n) - k + n = n(m+1) - k$.



Ejemplo 62. Considera la expresión $16(a^{10} - 18a) + 4$, en donde a es un entero positivo dado. Como el valor de a, y por tanto la salida, pueden ser muy grandes, obtén su congruencia módulo 11.

> **Solución.** Aplicaremos módulo a cada operación aritmética que se realiza, con la precaución de sumar 11 al momento de realizar la resta para evitar obtener residuos negativos.

Para obtener a^{10} escribiremos una función que obtenga la potencia de cualquier entero, misma que se devolverá modulada.

```
int potencia_modulo(int n, int pot, int mod)
{
    int resp = 1, i;
    for (i = 0; i < pot; i = i + 1)
        resp = resp * n % mod;
    return resp;
}

int solucion()
{
    int a, mod = 11, respuesta;
    cin>>a;

    //Resolviendo expresión por partes.
    respuesta = potencia_modulo(a, 10, mod);
    respuesta = respuesta - (18 % mod) * (a % mod) % mod + mod) % mod;
    respuesta = (((16 % mod) * respuesta) % mod + 4) % mod;

    return respuesta;
}
```

Ejemplo 63. Sean a y k números enteros. Si $6 \mid k$, obtén los últimos dos dígitos de la expresión:

$$\left(\left(a^k - 1\right)^{k/2} - 1\right)^{k/3} - 1$$

Solución. Dado que k es múltiplo de 6, los exponentes $k/2$ y $k/3$ serán enteros, por lo que las potencias involucradas también lo serán.

Obtener los dos últimos dígito de la expresión es equivalente a obtener su residuo módulo 100, no obstante habrá que considerar tres casos distintos:

- Si el residuo es 0, imprime “00”.
- Si el residuo es mayor que 0 y menor que 10 (se compone de un dígito), imprime el concatenado de “0” y el residuo.

- Si el residuo es mayor que 9 (se compone de dos dígitos), imprime el residuo mismo.

Recordemos que habrá que sumar el valor del módulo (100) para evitar residuos negativos.

```
void solucion()
{
    int a, k, mod = 100, residuo;
    cin>>a>>k;

    //Resolviendo la expresión por partes.
    residuo = (potencia_modulo(a, k, mod) - 1 + mod) % mod;
    residuo = (potencia_modulo(residuo, k/2, mod) - 1 + mod) % mod;
    residuo = (potencia_modulo(residuo, k/3, mod) - 1 + mod) % mod;

    if (residuo == 0)
        cout<<"00\n";
    else if (residuo < 10)
        cout<<"0"<<residuo<<"\n";
    else cout<<residuo<<"\n";
}
```

Ejemplo 64. Considera el resultado de la expresión $n! - n + 3$ para algún número natural n. Sumarás sus dígitos, y si lo que resulte es un número mayor a 9, sumarás los dígitos del resultado. Repetirás este proceso hasta que obtengas un único dígito, mismo que recibe el nombre de *raíz digital*. Devuelve dicho dígito.

Solución. Realizar la simulación del proceso como se describe en el problema no es eficiente, pues no se sabe cuántas veces será necesario sumar los dígitos de un número, así que buscaremos una solución que no requiera esta repetición. Llamemos A al resultado de $n!-n+3$. Sea $a_k a_{k-1} \ldots a_2 a_1 a_0$ la representación en dígitos de A, mismo que se puede desarrollar como sigue:

$$a_k a_{k-1} \ldots a_2 a_1 a_0 = a_k(10)^k + a_{k-1}(10)^{k-1} + \cdots + a_2(10)^2 + a_1(10) + a_0(1)$$

Nota que el primer paso para obtener la respuesta sería realizar la operación $a_k + a_{k-1} + \cdots + a_2 + a_1 + a_0$, así que lo que se necesita para pasar de la notación desarrollada de A a la suma de sus dígitos es encontrar la

forma de "neutralizar" las potencias de 10. El neutro multiplicativo de los enteros es 1, y si encontramos la manera de escribir cada potencia de 10 como un neutro multiplicativo, todas esas potencias adoptarían el valor de 1.

Pensemos en los posibles resultados de aplicarle a una potencia de 10 módulo 9: Ya que $10 \equiv 1(mod\ 9)$, para cualquier entero i se cumplirá que $10^i \equiv 1^i \equiv 1(mod\ 9)$; por otro lado, aplicarle módulo 9 a cualquier entero resultará en algún elemento del conjunto {0, 1, 2, 3, 4, 5, 6, 7, 8}, mismo que contiene ocho de las nueve posibles respuestas al problema.

Como n es un número natural, el menor valor posible de A es 3, por lo que no es posible que la suma de sus dígitos sea 0, no obstante, si un número es congruente con 0 módulo 9, éste es un múltiplo de 9, y el único dígito divisible por 9 (distinto de 0) es el mismo 9. Con esto excluimos la posibilidad de que la suma de los dígitos sea 0 y consideramos el escenario en el que la raíz digital sea 9.

Por lo que hemos señalado anteriormente, parece buena idea aplicar módulo 9 a la notación desarrollada de A.

$$a_k(10)^k + a_{k-1}(10)^{k-1} + \cdots + a_2(10)^2 + a_1(10) + a_0(1)\ (mod\ 9)$$

$$\begin{aligned}\equiv & a_k(10)^k\ (mod\ 9) + a_{k-1}(10)^{k-1}\ (mod\ 9) + \cdots + a_2(10)^2\ (mod\ 9) \\ & + a_1(10)\ (mod\ 9) + a_0(1)\ (mod\ 9)\end{aligned}$$

$$\equiv a_k(1)^k + a_{k-1}(1)^{k-1} + \cdots + a_2(1)^2 + a_1(1) + a_1(1)^0\ (mod\ 9)$$

$$\equiv a_k + a_{k-1} + \cdots + a_2 + a_1 + a_0\ (mod\ 9)$$

Con el desarrollo anterior probamos que todo número natural es congruente con la suma de sus dígitos módulo 9, por lo que aplicar módulo 9 a A será suficiente para resolver este ejercicio.

¿Es necesario obtener el número A para aplicar módulo 9? No, y tampoco es recomendable, pues a medida que crezca n será más probable que A supere los $9 \cdot 10^{18}$. Así que ni siquiera será necesario obtener A, simplemente aplicaremos módulo 9 a cada operación realizada en la expresión $n!-n+3$, recordando que si el resultado es 0, deberá ser 9 la salida.

Nota que codificaremos una función que calcule el factorial de hasta 10^3-1 y la utilizaremos como un preproceso, que nos otorga la ventaja de que cada consulta sea constante.

```
int f_modulo[1000];
void factorial_modulo(int mod)
{
    int i;
    f_modulo[0] = 1;
    for (i = 1; i < 1000; i = i + 1)
        f_modulo[i] = f_modulo[i - 1] * i % mod;
}

int solucion()
{
    int n, mod = 9, raiz_digital;
    cin>>n;
    factorial_modulo(mod);

    raiz_digital = (f_modulo[n] - n + mod + 3) % mod;

    if (raiz_digital == 0)
        raiz_digital = 9;

    return raiz_digital;
}
```

Ejemplo 65. Existe un algoritmo nombrado *Congruencial multiplicativo* para obtener números pseudoaleatorios. Que sean pseudoaleatorios implica que en algún punto se repetirán. Este método obedece a la siguiente ecuación, en donde a es un entero, n es un natural y x_0 es un natural nombrado *semilla* del que se alimentará la fórmula recursiva.

$$x_{i+1} = a \cdot x_i \ (mod\ n)$$

En el caso del congruencial multiplicativo, se han establecido condiciones para asegurar un periodo lo más largo posible, esto es, que se pueda obtener una gran cantidad de números aleatorios antes de que haya una repetición. No obstante, nosotros ignoraremos esas reglas y nos concentraremos en encontrar el tamaño del periodo dada la semilla x_0 y el entero n.

Para obtener el valor de a, encuentra la raíz digital de x_0 (vista en el ejercicio anterior).

> **Solución.** Por lo que vimos en el problema anterior, $a = x_0 \ \% \ 9$. Teniendo este dato, lo que resta es proceder la obtención de tantos x_i's como sean necesarios para encontrar al primer elemento repetido, pues ése será el indicativo de que el ciclo de números aleatorios volvió a comenzar.

```
1  int solucion()
2  {
3      int x_i, n, a, i;
4      int periodo[10000];
5      memset(periodo, 0, sizeof periodo)
6      cin>>n>>x_i;
7
8      a = x_i % 9;
9      for (i = 1; ; i = i + 1)
10     {
11         x_i = a * x_i % n;
12         if (periodo[x_i])
13             return i - periodo[x_i];
14
15         periodo[x_i] = i;
16     }
17 }
```

Ejemplo 66. En este ejercicio aprenderemos un poco de criptografía abordando el llamado *Cifrado de Vigenère*, que consiste en lo siguiente:

1. A cada letra del alfabeto le asignamos un entero no negativo comenzando por el 0: a=0, b=1, c=2, . . . , y=24, z=25.

2. Al texto que deseamos cifrar lo nombraremos “texto plano”, y tendremos una frase “llave” que concatenaremos hasta que se obtenga una cadena de la misma longitud que el texto plano. Tanto en el texto plano como en la frase llave se deben omitir los espacios y signos de puntuación para realizar este paso.

3. Letra a letra, se suman los equivalentes numéricos del texto plano y la cadena llave, y a cada suma se le aplica módulo 26 (el tamaño del alfabeto) para obtener un entero entre 0 y 25, que se sustituye por la letra que corresponde a dicho número.

Como entrada, tendrás una frase cifrada por este método, así como una frase llave. Tu trabajo será encontrar el texto plano.

Solución. Considera a p_i el i-éstimo caracter del texto plano (sin espacios ni signos de puntuación), a k_i el i-éstimo caracter de la frase llave concatenada y a c_i el i-ésimo caracter del texto cifrado.

Según el cifrado de Vigenère, el i-ésimo caracter del texto cifrado se obtiene de las operaciones $c_i = (p_i + k_i) \,\%\, 26$. La misión será, a partir de esa expresión, obtener el valor de p_i.

La sentencia $c_i = (p_i + k_i) \,\%\, 26$ es equivalente a la expresión matemática $p_i + k_i \equiv c_i \ (mod\ 26)$. De aquí, $p_i + k_i = (26x_i) + c_i$, donde $x_i \in \mathbb{Z}$. Despejando, $p_i = (26x_i) + c_i - k_i$; se sigue que $p_i \equiv c_i - k_i \ (mod\ 26)$. Aunado a lo anterior, hay que recordar que, cuando haya una resta involucrada, habrá que sumar 26 para evitar la obtención de residuos negativos, con lo que llegamos a que, para obtener p_i, deberemos aplicar las operaciones $p_i = [(c_i - k_i) + 26] \,\%\, 26$ y devolver la letra que corresponda al resultado.

Como último comentario, por comodidad, en lugar de concatenar K (la llave) hasta que alcance la longitud de P (el texto plano), cada t veces usaremos la i-ésima letra de K, como se muestra en la línea 18 del programa solución.

```
1  string solucion()
2  {
3      string C, P, K;
```

```
4       int i, j;
5       cin>>C>>K;
6
7       P.resize(C.size());
8       for (i = j = 0; i < C.size(); i = i + 1, j = j + 1)
9       {
10          //Si no es una letra:
11          if (not isalpha(C[i]))
12          {
13              P[i] = C[i];
14              j = j - 1;
15          }
16          else
17          {
18              P[i] = (C[i] - K[j % K.size()] + 26) % 26;
19              P[i] = P[i] + 'a'; //Convirtiendo a caracter.
20          }
21      }
22
23      return P;
24  }
```

El ejemplo anterior es una muestra de uno de los métodos de cifrado más sencillos, pero si profundizas un poco más verás lo complejas que pueden llegar a ser las técnicas de criptografía y el importante papel que juega la teoría de números en ellos. En [9] y [12] puedes comenzar tu exploración por el mundo de la criptografía, y en [13] y [14] tendrás la oportunidad de ampliarla a métodos más sofisticados con una mayor profundidad.

2.3.1. Divisiones en Aritmética Modular

Como habrás notado, no hay una regla que mantenga la relación modular involucrando la división. Esto se debe a que, trabajando con los residuos de la división, no es posible asegurar que dividirlos resultará en un número entero, y hay que recordar que los módulos únicamente se mueven sobre los enteros. Además, aun cuando la división

resulte en un entero, no podemos asegurar que al aplicarle el módulo, éste sea equivalente a la modulación de cada número de la división. Por ejemplo, $12 \equiv 0 \ (mod\ 12)$, pero si dividimos ambos lados de la congruencia entre 4, la relación no se mantiene, pues llegaríamos a la expresión $4 \equiv 0 \ (mod\ 12)$, lo que claramente es incorrecto.

De inicio, es cierto que no hay una regla general que ayude a conservar una relación modular cuando tenemos divisiones, sin embargo, es posible asegurar la congruencia de una división módulo n si se cumplen las condiciones que se enuncian en la siguiente proposición.

Proposición 12. Sean a, b y k números enteros, con $k \neq 0$. Si $k \Big| a$, $k \Big| b$, y para algún natural n que no tiene divisores en común con k (salvo ± 1), se cumple que $a \equiv b \ (mod\ n)$, entonces es cierto que:

$$\frac{a}{k} \equiv \frac{b}{k} \ (mod\ n)$$

Demostración. De $k \Big| a$ y $k \Big| b$, se sabe que $k \Big| a - b$, y de $a \equiv b \ (mod\ n)$, se sabe que $n \Big| a - b$. Por tanto, $a - b$ puede ser escrito como $a - b = n \cdot x$ para algún entero x. Se sigue que $k \Big| n \cdot x$.

Como k y n no tienen factores en común, podemos asegurar que k no divide a n, y por tanto, debe dividir a x. De ahí, existe un entero y que cumple $x = k \cdot y$. Luego, $a - b = n \cdot k \cdot y$. Despejando:

$$\frac{a - b}{k} = n \cdot y$$

De lo que se deduce que el resultado de la división es entero, más aún, que $n \Big| \frac{a}{k} - \frac{b}{k}$, y por tanto, que $\frac{a}{k} \equiv \frac{b}{k} \ (mod\ n)$.

□

Ejemplo 67. Según la proposición que acabamos de revisar, determina para qué entradas a, b, k (enteros) y n (natural), se cumplirá que $\frac{a}{k} \equiv \frac{b}{k} \ (mod\ n)$.

> **Solución.** Lo primero que hay que revisar es que k sea distinto de 0, si no se cumple, se concluye que la relación de congruencia tampoco se cumple.

En caso contrario, procedemos a la siguiente prueba, que consistirá en asegurarse de que k divida a a y a b, lo que en código sería equivalente a que se cumpla que $a \,\%\, k = 0$ y $b \,\%\, k = 0$.

Luego, veremos si $a \equiv b \ (mod\ n)$. Nuevamente, si esto no se satisface, concluimos que no se puede asegurar la relación, de otro modo continuamos con la última validación.

La última validación, que consiste en verificar que n y k no tengan factores en común (salvo ± 1), requerirá más tiempo que las anteriores. Más adelante, en la sección de *Máximo Común Divisor*, veremos un método más eficiente, no obstante, con lo que hemos estudiado hasta ahora, sólo nos alcanza para una búsqueda entre todos los divisores de n y k, que, por la proposición 8, se puede reducir hasta la raíz cuadrada de alguno de ellos.

```
1  bool solucion()
2  {
3      int a, b, n, k, i, raiz_n;
4      bool se_cumple = true;
5      cin>>a>>b>>n>>k;
6
7      if (k == 0)
8          se_cumple = false;
9      else if ( not (a % k == 0 and b % k == 0) )
10         se_cumple = false;
11     else if ( not (a % n == b % n) )
12         se cumple = false;
13     else
14     {
15         raiz_n = sqrt(n);
16         for (i = 2; i <= raiz_n; i = i + 1)
17             if (k % i == 0 and n % i == 0)
18                 se_cumple = false;
19     }
20
21     return se_cumple;
22 }
```

Ejemplo 68. Se está planificando un evento de atletismo. Una de las pruebas propuestas se compone de dos carreras de distinta longitud m_1 y m_2, que tendrán lugar en una pista de n metros planos, en donde n es un número primo.

Los organizadores requieren que se cumplan ciertas condiciones en las carreras. Para eso, estimaron una longitud promedio k de una zancada (entero primo distinto de n), y han pedido que se asegure que ambas rondas en la prueba se puedan recorrer con zancadas completas de longitud k; también requieren que la diferencia entre el número de zancadas dadas en la primera y la segunda carrera sea múltiplo de n; finalmente, desean que ambas pruebas comiencen en el mismo punto y finalicen en el mismo punto (pero los puntos de inicio y final no necesariamente son iguales entre sí).

Si los organizadores ya han establecido los valores de m_2, n y k, y se han asegurado de que se respeten las condiciones pedidas (las posibles teniendo únicamente esos datos), encuentra cuántas posibles longitudes puede tomar m_1, considerando que la segunda carrera (m_2) debe ser más larga que la primera (m_1).

> **Solución.** Asegurar que ambas rondas se recorran en zancadas completas es equivalente a decir que buscamos que $k \Big| m_1$ y $k \Big| m_2$, aunque nosotros únicamente nos preocuparemos por que $k \Big| m_1$, pues está dicho que los organizadores se han encargado de m_2.
>
> Si m_1 y m_2 son las longitudes de las carreras y k es la distancia en la que se estima una zancada, el número de zancadas que se dan en cada prueba es m_1/k y m_2/k, respectivamente. Así, si queremos que la diferencia entre el número de zancadas sea múltiplo de n, simbólicamente lo que deseamos es que $n \Big| \frac{m_1}{k} - \frac{m_2}{k}$. Esto, a su vez, es equivalente a que $\frac{m_1}{k} \equiv \frac{m_2}{k} \ (mod\ n)$. Para llegar a este punto, buscaremos probar que se cumplan las condiciones enunciadas en la proposición 12.
>
> Ya que ambas pruebas comienzan en el mismo punto y terminan en el mismo punto, la longitud de éstas pueden ser escritas como $m_1 = n \cdot v_1 + r$ y $m_2 = n \cdot v_2 + r$, en donde v_1 y v_2 son el número de vueltas enteras a la pista, y pueden ser incluso 0. Además, podemos asegurar que $v_2 > v_1$ si queremos que $m_2 > m_1$. Dado que ambas longitudes tienen el mismo residuo al dividirlo entre n, se sigue que $m_1 \equiv m_2 \ (mod\ n)$.

Resumiendo los puntos anteriores, para encontrar los posibles valores de m_1, habrá que buscar los múltiplos de k menores que m_2, esto es, los números de la forma $k \cdot i$, en donde $0 \leq i < M$ y $M = m_2/k$; para cada número que cumpla esa condición, habrá que validar que $m_1 \,\%\, n = m_2 \,\%\, n$; si lo anterior se satisface, por la proposición 12 y dado que n y k son primos, y por tanto, no tienen factores en común (salvo ± 1), la condición de que $\frac{m_1}{k} \equiv \frac{m_2}{k}\ (mod\ n)$ no se tiene que comprobar, pues podemos asegurar que es cierto.

```
1  int solucion()
2  {
3      int k, m2, n, i, M, valores_m1 = 0;
4      cin>>m2>>n>>k;
5
6      M = m2 / k;
7      for (i = 0; i < M; i = i + 1)
8          if (k * i % n == m2 % n)
9              valores_m1 = valores_m1 + 1;
10
11     return valores_m1;
12 }
```

Teorema 11. [Pequeño Teorema de Fermat]. Sea p un número primo. Para todo entero a que no sea divisible por p se cumplirá que:

$$a^{p-1} \equiv 1\ (mod\ p)$$

Demostración. Considera el conjunto $S = \{1, 2, 3, \ldots, p-1\}$; p no divide a ningún elemento de ese conjunto, pues es mayor que todos ellos, y ya que p tampoco divide a a, podemos asegurar que p no es divisor de ningún elemento del conjunto $S_a = \{a, 2a, 3a, \ldots, (p-1)a\}$, lo que, en aritmética modular, implica que ningún elemento de S_a cumple que $i \cdot a \equiv 0\ (mod\ p)$, $1 \leq i \leq p-1$. Así, si a cada elemento de S_a le aplicamos módulo p, todos arrojarán un natural entre 1 y $p-1$, inclusive. La pregunta ahora es cuáles de esos valores se obtendrán de dicha acción.

Tomemos dos enteros del conjunto S_a que sean congruentes módulo p, digamos $i_1 \cdot a$ y $i_2 \cdot a$. Ya que $i_1 \cdot a \equiv i_2 \cdot a\ (mod\ p)$, se sigue que $p \Big| (i_1 \cdot a) - (i_2 \cdot a)$, y por tanto, $p \Big| a(i_1 - i_2)$. Como $p \not| a$, se tiene que cumplir que $p \Big| (i_1 - i_2)$; además, como $1 \leq i_1, i_2 \leq p-1$,

sabemos que la diferencia en valor absoluto entre i_1 e i_2 también es menor que $p-1$, es decir, $|i_1 - i_2| < p - 1$.

Para saber qué valores podrían adoptar i_1 e i_2, veamos cuánto podría valer $|i_1 - i_2|$: Cuando buscamos $|i_1 - i_2|$, buscamos un entero no negativo menor que $p-1$ que sea divisible por p, y el único número que encaja con esa descripción es 0, de lo que se concluye que $i_1 = i_2$, es decir, que $i_1 \cdot a = i_2 \cdot a$. De aquí, no existe forma de que haya dos elementos distintos de S_a que sean congruentes módulo p. Así, dado que todos los residuos módulo p del conjunto S_a son distintos y sólo pueden tomar valores enteros entre 1 y $p-1$, podemos asegurar que, en S_a, hay exactamente un elemento congruente con $1, 2, 3, \ldots, p-1$.

Luego, $a(2a)(3a)\cdots[(p-1)a] \equiv 1(2)(3)\cdots(p-1)\ (mod\ p)$

$$\implies (p-1)! \cdot a^{p-1} \equiv (p-1)!\ (mod\ p)$$

Con base en la equivalencia anterior, ¿podemos asegurar que $a^{p-1} \equiv 1\ (mod\ p)$, o lo que es lo mismo, que $\frac{(p-1)!\cdot a^{p-1}}{(p-1)!} \equiv \frac{(p-1)!}{(p-1)!}\ (mod\ p)$? Veamos si se cumplen las condiciones enunciadas en la proposición 12 para responder esa pregunta.

Nota que $(p-1)!$ divide tanto a $a^{p-1}(p-1)!$ como a $(p-1)!$; $(p-1)!$ y p no tienen factores en común (salvo 1), pues los únicos divisores de p son 1 y p mismo, y p no divide al producto $1(2)(3)\cdots(p-1)$. Por tanto, podemos aplicar la proposición 12 y concluir que, dado que $(p-1)! \cdot a^{p-1} \equiv (p-1)!\ (mod\ p)$,

$$a^{p-1} \equiv 1\ (mod\ p)$$

Demostrando así el teorema.

□

Ejemplo 69. Para dos números naturales L y k tales que $L \geq k$, encuentra todos los primos impares p menores que L que satisfagan:

$$p \Big| 2^p + k$$

Solución. El Pequeño Teorema de Fermat nos dice que, si $p \not| 2$, podemos asegurar que $2^{p-1} \equiv 1(mod\ p)$, y ya que p y 2 son primos, 2 no puede

ser divisible por p. Por otro lado, cuando probamos que los módulos resultaban en una relación de equivalencia (en el teorema 9), vimos que podíamos establecer que $2 \equiv 2\ (mod\ p)$. Multiplicando las dos expresiones anteriores, $2^p \equiv 2\ (mod\ p)$, de lo que se deduce que $p \Big| 2^p - 2$.

Luego, $2^p + k = (2^p - 2) + (k + 2)$. Si ya tenemos conocimiento de que $p \Big| 2^p - 2$, para que $p \Big| 2^p + k$, únicamente falta asegurar que $p \Big| k + 2$. Así, sólo tenemos que buscar a los primos que dividan a $k + 2$, evitando con ello el proceso de la potencia 2^p; más aún, no será necesario buscar hasta L, sino únicamente desde 3 hasta $k + 2$, pues después de ese entero, no habrá natural que lo pueda dividir.

Utilizaremos la criba de Eratóstenes para obtener los primos menores o iguales que $k + 2$.

```
1  vector<int> solucion()
2  {
3      int L, k, i;
4      vector<int> divisores;
5      cin>>L>>k;
6
7      criba(k + 2);
8
9      vector<int> divisores;
10     for (i = 1; primos[i] <= k + 2; i = i + 1 )
11         if ((k + 2) % primos[i] == 0)
12             divisores.push_back(primos[i]);
13
14     return divisores;
15 }
```

El Pequeño Teorema de Fermat nos brinda, además, un par de métodos más para realizar divisiones, mismos que analizaremos a continuación. Lo que tienen en común estas técnicas es que, en realidad en ninguna de las dos se realizan divisiones, sino que éstas son sustituidas por expresiones equivalentes, modularmente hablando.

Método 1

Proposición 13. Sea p un número primo y a un entero tal que $p \nmid a$. Para cualquier entero n múltiplo de a se satisfacerá:

$$\frac{n}{a} \equiv n \cdot a^{p-2} \ (mod\ p)$$

Demostración. Del Pequeño Teorema de Fermat, sabemos que $a^{p-1} \equiv 1 \ (mod\ p)$; por otro lado, $\frac{n}{a} \equiv \frac{n}{a} \ (mod\ p)$. Aprovechando las operaciones permitidas en aritmética modular, podemos deducir que:

$$a^{p-1} \cdot \frac{n}{a} \equiv 1 \cdot \frac{n}{a} \ (mod\ p)$$

$$\implies a^{p-2} \cdot n \equiv \frac{n}{a} \ (mod\ p)$$

$$\implies \frac{n}{a} \equiv a^{p-2} \cdot n \ (mod\ p)$$

□

Así pues, concluimos que la división n/a es equivalente a $a^{p-2} \cdot n$, módulo p, y ya que esta última expresión no incluye ninguna división, no habrá ninguna complicación al aplicarle módulos.

La sentencia que corresponde a la expresión obtenida, que además asegura que el resultado estará dentro de $[0, p-1]$ es:

$$((a^{p-2} \,\%\, p) \cdot (n \,\%\, p)) \,\%\, p$$

Recuerda que, a cada paso, aplicamos módulo p para evitar un desbordamiento en los números obtenidos.

Todas las operaciones que se incluyen en la sentencia de arriba son constantes, salvo la potencia que resulta más tardada; si bien la primera forma de programarla que

viene a la mente es multiplicar a $p-2$ veces, ya sea de manera iterativa o de manera recursiva, este método tendría una complejidad lineal (pues realizaría una operación $p-2$ veces), sin embargo, más adelante estudiaremos una técnica que permite realizar la potenciación de un número en tiempo logarítmico.

Ejemplo 70. Dado un arreglo de enteros de tamaño 10^5, determina el producto, módulo 71, de los elementos en un intervalo señalado $[a, b]$. Por ejemplo, dado el arreglo $< 3, 2, -8, -1, 2 >$, el producto en el intervalo $[1, 3]$ es $2(-8)(-1) = 16$, que permanece de la misma manera al aplicarle módulo 71.

Solución. La solución obvia es tomar todos los elementos del intervalo $[a, b]$ y multiplicarlos al tiempo que aplicamos módulos a los enteros obtenidos, lo que implica un algoritmo de complejidad lineal, sin embargo, en programación competitiva casi nunca es tan sencillo. Si tu programa tuviera que realizar más de 10^3 consultas, seguramente el veredicto será *Límite de tiempo*, así que busquemos una solución que funcione incluso para esta posibilidad.

Calculemos el producto de todos los subconjuntos $[0, j]$, para $j = 0, 1, 2, \ldots, T-1$, en donde T es el tamaño del arreglo. Obtener el producto de un subarreglo $[i, j]$ será equivalente a dividir el producto de $[0, j]$ entre el producto de $[0, i-1]$, y ambos fueron obtenidos ya; además, podemos garantizar que el primer producto es divisible entre el segundo, pues comparten todos los factores del subarreglo $[0, i-1]$. De aquí, y por la proposición 13:

$$\begin{aligned} P_{a,b} \,\%\, 71 &= \frac{P_{0,b}}{P_{0,a-1}} \,\%\, 71 \\ &= \left[(P_{0,a-1})^{69} \cdot P_{0,b} \right] \%\, 71 \\ &= \left[(P_{0,a-1})^{69} \,\%\, 71) \cdot (P_{0,b} \,\%\, 71) \right] \%\, 71 \end{aligned}$$

En donde $P_{i,j}$ representa el producto de los elementos del subarreglo $[i, j]$.

Puede parecer que con la expresión obtenida terminamos, pero hay un

caso en el que se debe tener especial cuidado: Si $a = 0$, no tendremos un producto por el cual dividir, pues no existirá un elemento $a - 1$. Esto se resolverá separando la respuesta en dos casos: Si $a = 0$, devolveremos únicamente $P_{0,b} \,\%\, 71$; de lo contrario, devolveremos la fórmula a la que llegamos arriba.

```
#define tam 100000
int arreglo[tam], producto[tam], mod = 71;

void producto_arreglo_modulo()
{
    int i;
    producto[0] = arreglo[0] % mod;
    for (i = 1; i < tam; i = i + 1)
        producto[i] = producto[i - 1] * arreglo[i] % mod;
}

int solucion()
{
    int a, b, respuesta;
    cin>>a>>b;

    if (a == 0)
        return producto[b];
    else
    {
        respuesta = potencia_modulo(producto[a-1], mod-2, mod);
        respuesta = respuesta * producto[b] % mod;

        return respuesta;
    }
}
```

Ejemplo 71. Dado un número natural n, devuelve el resultado de la siguiente expresión después de aplicarle módulo 17.

$$\sum_{i=0}^{n} \binom{n}{i}$$

Solución. Aunque en el capítulo de *Análisis Combinatorio* se presentó una forma de reducir la suma pedida a una expresión de complejidad

constante, en este problema la resolveremos sumando por sumando para poner en práctica el método recientemente visto.

Recordemos que:

$$\binom{n}{i} = \frac{n!}{i! \cdot (n-i)!}$$

Aplicar el método 1 a cada coeficiente binomial luciría como sigue:

$$\frac{n!}{i! \cdot (n-i)!} \,\%\, 17 = \Big[(i! \cdot (n-i)!)^{15} \cdot n!\Big] \,\%\, 17$$

$$= \Big[\Big[(i! \cdot (n-i)!)^{15}\Big] \,\%\, 17\Big] \cdot \Big[n! \,\%\, 17\Big]\Big] \,\%\, 17$$

Finalmente, hay que considerar que, al agregar cada sumando, será necesario aplicar módulo también; utilizaremos la función $factorial\_modulo$ como un preproceso, con lo que dispondremos del vector f_modulo para acceder al factorial de i, módulo 17.

```
1  int solucion()
2  {
3      int sumando, suma = 0, i, n, mod = 17;
4      cin>>n;
5
6      for (i = 0; i <= n; i = i + 1)
7      {
8          sumando = potencia_modulo(f_modulo[i] * f_modulo[n-i], mod-2,
9                    mod) * (f_modulo[n] % mod) % mod;
10         suma = (suma + sumando) % mod;
11     }
12
13     return suma;
14 }
```

Como ya dijimos y fue probado en este programa, es posible volver el funcionamiento del proceso más eficiente si, en un preproceso, calculamos todos los factoriales de cada entero no negativo menor o igual que algún entero M muy grande, mismo que deberás elegir tú. Podrás encontrar dicho preproceso en el ejemplo 64.

Método 2: Factorial Reverso

En el capítulo de *Análisis Combinatorio*, esclarecimos que el máximo coeficiente binomial que es posible resolver en C++ es $\binom{66}{33}$, no obstante, podemos obtener una equivalencia modular de dicha expresión. El problema radica entonces en que la resolución de un coeficiente binomial implica una división, lo que complica la obtención de su representación modular.

El método que veremos aquí sirve justamente para resolver este conflicto, de hecho, será útil para cualquier división que resulte en un entero en la que el divisor se pueda escribir como un factorial, o como un producto de factoriales.

Para obtener el factorial reverso de una división de este tipo, será necesaria *tu* elección de algún entero positivo M, cuyas características se enunciarán en las proposiciones a continuación.

Proposición 14. Sea p un número primo y M un número natural tal que $M < p$. Si la división $n/k!$ resulta en un entero para $n, k \in \mathbb{Z}$ y $k \leq M$, decimos que el *factorial reverso* de $\frac{n}{k!}$, con el que, además, es congruente módulo p es:

$$(M!)^{p-2} \cdot n \cdot \Big[M(M-1)(M-2)\cdots(k+1)\Big] \ (mod\ p)$$

Nota: Aunque, estrictamente hablando, M debe ser mayor o igual que k, es recomendable que $M!$ también sea mayor o igual que n, para cubrir la posibilidad de que $k!$ deba tomar valores desde 0 (o cualquier otro entero) hasta n.

Demostración. Como p es un primo mayor que M, podemos asegurar que $p \nmid M!$, por lo que podemos aplicar el Pequeño Teorema de Fermat, que nos lleva a que $(M!)^{p-1} \equiv 1 \ (mod\ p)$. Por otro lado, $\frac{n}{k!} \equiv \frac{n}{k!} \ (mod\ p)$. Multiplicando estas dos expresiones, obtenemos que:

$$(M!)^{p-1} \cdot \frac{n}{k!} \equiv \frac{n}{k!} \ (mod\ p)$$

Concentrémonos, por el momento, en el lado izquierdo de la equivalencia, que puede ser reescrita como sigue.

$$(M!)^{p-1} \cdot \frac{n}{k!} = (M!)^{p-2} \cdot n \cdot \frac{M!}{k!}$$

$$= (M!)^{p-2} \cdot n \cdot \frac{M(M-1)(M-2)\cdots(k+1)\cancel{k!}}{\cancel{k!}}$$

$$= (M!)^{p-2} \cdot n \cdot \Big[M(M-1)(M-2)\cdots(k+1)\Big]$$

Sustituyendo en la congruencia modular:

$$(M!)^{p-2} \cdot n \cdot \Big[M(M-1)(M-2)\cdots(k+1)\Big] \equiv \frac{n}{k!} \ (mod\ p)$$

$$\implies \frac{n}{k!} \equiv (M!)^{p-2} \cdot n \cdot \Big[M(M-1)(M-2)\cdots(k+1)\Big] \ (mod\ p)$$

□

En la proposición anterior demostramos que podemos obtener una expresión modularmente equivalente a una división, siempre que el denominador sea un factorial. ¿Cómo se modifica el proceso si el denominador es el producto de varios factoriales?

Proposición 15. Sea p un número primo y M un número natural tal que $M < p$. Considera la expresión $\frac{n}{k_1! \cdot k_2!}$ que resulta en un entero y para la que $k_1, k_2 \leq M$. El *factorial reverso* de $\frac{n}{k_1! \cdot k_2!}$ es:

$$(M!)^{p-3} \cdot n \cdot \Big[M(M-1)(M-2)\cdots(k_1+1)\Big] \cdot \Big[M(M-1)(M-2)\cdots(k_2+1)\Big] \ (mod\ p)$$

Nota: Aunque estrictamente hablando, M debe ser mayor que todas las $k'i$ presentes en el divisor del problema, es recomendable que $M!$ también sea mayor que n, para cubrir la posibilidad de que alguna $k_i!$ deba tomar valores desde 0 (o cualquier otro entero) hasta n.

Demostración. Seguiremos un procedimiento muy similar al de la proposición 14.

De $(M!)^{p-1} \equiv 1 \ (mod\ p)$ y $\frac{n}{k_1! \cdot k_2!} \equiv \frac{n}{k_1! \cdot k_2!} \ (mod\ p)$, se deduce que:

$$(M!)^{p-1} \cdot \frac{n}{k_1! \cdot k_2!} \equiv \frac{n}{k_1! \cdot k_2!} \ (mod\ p)$$

Luego, fijándonos en el lado izquierdo de la congruencia:

$$\begin{aligned}
(M!)^{p-1} \cdot \frac{n}{k_1! \cdot k_2!} &= \frac{(M!)^{p-2} \cdot n}{k_2!} \cdot \frac{M!}{k_1!} \\
&= \frac{(M!)^{p-2} \cdot n}{k_2!} \cdot \frac{M(M-1)(M-2)\cdots(k_1+1)\ \cancel{k_1!}}{\cancel{k_1!}} \\
&= \frac{(M!)^{p-2} \cdot n}{k_2!} \cdot \Big[M(M-1)(M-2)\cdots(k_1+1)\Big] \\
&= (M!)^{p-3} \cdot n \cdot \frac{M!}{k_2!} \cdot \Big[M(M-1)\cdots(k_1+1)\Big] \\
&= (M!)^{p-3} \cdot n \cdot \frac{M(M-1)\cdots(k_2+1)\ \cancel{k_2!}}{\cancel{k_2!}} \cdot \Big[M(M-1)\cdots(k_1+1)\Big] \\
&= (M!)^{p-3} \cdot n \cdot \Big[M(M-1)\cdots(k_2+1)\Big] \cdot \Big[M(M-1)\cdots(k_1+1)\Big]
\end{aligned}$$

Así, concluimos que, ya que $\frac{n}{k_1! \cdot k_2!} \equiv (M!)^{p-1} \cdot \frac{n}{k_1! \cdot k_2!}\ (mod\ p)$,

$$\frac{n}{k_1! \cdot k_2!} \equiv (M!)^{p-3} \cdot n \cdot \Big[M(M-1)\cdots(k_1+1)\Big] \cdot \Big[M(M-1)\cdots(k_2+1)\Big]\ (mod\ p)$$

□

En este caso, para optimizar el procedimiento, necesitaremos tres preprocesos: Cada producto de la forma $M(M-1)\cdots(k_i+1)$, el factorial de M, y la potencia aplicada a $M!$ (que son convenientes porque M es constante). Si bien es cierto que la ejecución de un factorial reverso requiere más líneas de código para obtener esos

preprocesos, también es cierto que, de esta forma, cada consulta tendría una complejidad constante.

Recordemos nuevamente que $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, por lo que el factorial reverso es ideal para modular un coeficiente binomial (e incluso un coeficiente multinomial), mas esto no implica que no se pueda aplicar el *Método 1*, como se vio en el ejemplo 71.

$$\begin{aligned}\binom{n}{k} =& \frac{n!}{k!(n-k)!} \\ \equiv& (M!)^{p-3} \cdot n \cdot \Big[M(M-1)\cdots(k+1)\Big] \cdot \Big[M(M-1)\cdots(n-k+1)\Big] \ (mod\ p)\end{aligned}$$

El método anterior es propuesto por esta obra y en cada ejercicio que requiera de un procedimiento como el descrito arriba se apegará a él, sin embargo, es probable que hayas visto (o en algún momento encuentres) una versión de factorial reverso como la siguiente:

$$n \cdot \Big[(M!)^{p-2} \cdot (M)(M-1)\cdots(k_1+1)\Big] \cdot \Big[(M!)^{p-2} \cdot (M)(M-1)\cdots(k_2+1)\Big] \ (mod\ p)$$

Ambos métodos son correctos y tienen la misma complejidad; la diferencia radica en que en el método propuesto por esta obra se deberá obtener la potencia $p-x-1$ de $M!$, en donde x es el número de factoriales que conforman el divisor de la división en cuestión (es decir, el número de k_i's), mientras que con la otra técnica se tendrá que resolver la potencia $(M!)^{p-2}$ e invocarla tantas veces como k_i's haya.

Pese a no ser propia de esta obra, demostraremos que la segunda técnica mostrada también es útil para la obtención de la representación modular de una división.

Ya vimos que:

$$(M!)^{p-1} \cdot \frac{n}{k_1! \cdot k_2!} = \frac{(M!)^{p-2} \cdot n}{k_2!} \cdot \Big[M(M-1)\cdots(k_1+1)\Big]$$

Llamando nuevamente al Pequeño Teorema de Fermat, $(M!)^{p-1} \equiv 1\ (mod\ p)$; y

tomando en cuenta que todo entero es congruente consigo mismo sin importar el módulo, se deduce:

$$\frac{(M!)^{p-2}\cdot n}{k_2!}\cdot\Big[M(M-1)\cdots(k_1+1)\Big](M!)^{p-1} \equiv \frac{(M!)^{p-2}\cdot n}{k_2!}\cdot\Big[M(M-1)\cdots(k_1+1)\Big] \ (mod\ p)$$

Tomamos el lado izquierdo de la congruencia y la reescribimos:

$$\frac{(M!)^{p-2}\cdot n}{k_2!}\cdot\Big[M(M-1)\cdots(k_1+1)\Big](M!)^{p-1}$$

$$= n\cdot\Big[(M!)^{p-2}\cdot M(M-1)\cdots(k_1+1)\Big]\cdot\frac{(M!)^{p-1}}{k_2!}$$

$$= n\cdot\Big[(M!)^{p-2}\cdot M(M-1)\cdots(k_1+1)\Big]\cdot(M!)^{p-2}\cdot\frac{M!}{k_2!}$$

$$= n\cdot\Big[(M!)^{p-2}\cdot M(M-1)\cdots(k_1+1)\Big]\cdot(M!)^{p-2}\cdot\frac{M(M-1)\cdots(k_2+1)\,\cancel{k_2!}}{\cancel{k_2!}}$$

$$= n\cdot\Big[(M!)^{p-2}\cdot M(M-1)\cdots(k_1+1)\Big]\cdot\Big[(M!)^{p-2}\cdot M(M-1)\cdots(k_2+1)\Big]$$

Sustituyendo en la congruencia,

$$\frac{n}{k_1!\cdot k2!}$$

$$\equiv n\cdot\Big[(M!)^{p-2}\cdot M(M-1)\cdots(k_1+1)\Big]\cdot\Big[(M!)^{p-2}\cdot M(M-1)\cdots(k_2+1)\Big] \ (mod\ p)$$

Ejemplo 72. ¿Cuántas cadenas de tamaño k se pueden formar sin letras repetidas a partir del alfabeto latino moderno, en donde k es un entero positivo entre 2 y 26, inclusive? Devuelve la respuesta módulo 31.

Solución. La fórmula que cuenta las cadenas que buscamos es

$$\frac{26!}{k!}$$

Si tienes dudas de por qué esto es cierto, acude a la sección de *Permutaciones* en el capítulo de *Análisis Combinatorio*.

Ya que tenemos una fórmula, enfoquémonos en aplicarle módulo 31 a través de un factorial reverso. Para ello, estableceremos el valor de M en 27 (pues k puede valer hasta 26); recuerda que el valor de M será tu elección, pero debe satisfacer las condiciones enunciadas en la proposición 14. Así:

$$\frac{26!}{k!} \,\%\, 31 = (27!)^{29} \cdot 26! \cdot \Big[27(26)(25)\cdots(k+1)\Big] \,\%\, 31$$

$$= \Big[\Big[(27!)^{29} \,\%\, 31\Big] \cdot \Big[26! \,\%\, 31\Big] \cdot \Big[27(26)\cdots(k+1) \,\%\, 31\Big]\Big] \,\%\, 31$$

Nuevamente, haremos uso de un preproceso que obtenga los factoriales modulados necesarios, mismos a los que accederemos en el arreglo *f_modulo*. Requeriremos también la creación de una función que obtenga el producto $M(M-1)(M-2)\cdots(k+1)$, a la que nombraremos *llenar_PReverso*.

```
#define M 27
int prod_reverso[M + 1];

void llenar_PReverso(int mod)
{
    int k;
    prod_reverso[M] = 1;
    for (k = M - 1; k >= 0; k = k - 1)
        prod_reverso[k] = prod_reverso[k + 1] * (k + 1) % mod;
}

int solucion()
{
    int respuesta, mod = 31, k;
    cin>>k;
```

```
16
17      //Resolviendo la expresión por partes.
18      respuesta = potencia_modulo(f_modulo[27], mod-2, mod);
19      respuesta = respuesta * f_modulo[26] % mod;
20      respuesta = respuesta * prod_reverso[k] % mod;
21
22      return respuesta;
23  }
```

Ejemplo 73. Dado un número natural n menor que 100, aplica factorial reverso módulo 127 a la siguiente expresión.

$$\sum_{i=0}^{n} \binom{n}{i}$$

Solución. Establezcamos un M menor que 127 y mayor o igual que 100, digamos $M = 101$.

Cada coeficiente binomial tiene un producto de exactamente dos factoriales en el denominador, por tanto, el exponente que corresponderá a $M!$ será $p - 1 - 2 = 124$. Luego,

$$\frac{n!}{i! \cdot (n-i)!} \,\%\, 127$$

$$= \Big[(101!)^{124} \cdot n! \cdot \big[101(100)\cdots(i+1)\big] \cdot \big[101(100)\cdots(n-i+1)\big] \Big] \,\%\, 127$$

$$= \Big[\big[(101!)^{124} \,\%\, 127\big] \cdot \big[n! \,\%\, 127\big] \cdot \big[101(100)\cdots(i+1) \,\%\, 127\big] \cdot \big[101(100)\cdots(n-i+1) \,\%\, 127\big] \Big] \,\%\, 127$$

Finalmente, aplicamos módulo después de agregar cada sumando, como se indica en el programa a continuación.

```
1   int solucion()
2   {
```

```
    int suma = 0, sumando, i, n, mod = 127, M = 101;
    int M_fact_pot = potencia_modulo(f_modulo[M], mod-3, mod);
    cin>>n;

    for (i = 0; i <= n; i = i + 1)
    {
        //Resolviendo cada sumando por partes.
        sumando = M_fact_pot * f_modulo[n] % mod;
        sumando = sumando * prod_reverso[i] % mod;
        sumando = sumando * prod_reverso[n - i] % mod;

        suma = suma + sumando % mod;
    }

    return suma;
}
```

Ejemplo 74. Sea n un número natural mayor que 2 y menor que 100 y k algún entero no negativo menor que $\lfloor n/3 \rfloor$. Considera la expresión:

$$\binom{n}{k, 2k, x} - 1$$

Determina todos los posibles valores de x y representa todos los posibles valores de la expresión dada, módulo 131.

Solución. En el capítulo de *Análisis Combinatorio* vimos que, por ser un coeficiente multinomial, debe ocurrir que $k + 2k + x = n$, de lo que se deduce que el único valor posible para x será $n - 3k$, por tanto, la expresión dada únicamente podrá adoptar un valor también.

Recordemos que:

$$\binom{n}{k, 2k, n-3k} = \frac{n!}{k! \cdot (2k)! \cdot (n-3k)!}$$

Para aplicar módulos a esta división, consideremos a $M = 100$, mismo que, debido a que en el denominador tenemos un producto de tres factoriales, deberá elevarse a la potencia $p - 1 - 3 = p - 4 = 127$. Entonces:

$$\frac{n!}{k! \cdot (2k)! \cdot (n-3k)!} - 1 \,\%\, 131$$

$$= \Bigg[\Big[(100!)^{127} \cdot n! \cdot \Big[100(99)(98) \cdots (k+1) \Big] \cdot \Big[100(99)(98) \cdots (2k+1) \Big] \cdot \Big[100(99)(98) \cdots (n-3k+1) \Big] \Big] - 1 \Bigg] \,\%\, 131$$

$$= \Bigg[\Big[\Big[(100!)^{127} \,\%\, 131 \Big] \cdot \Big[n! \,\%\, 131 \Big] \cdot \Big[100(99) \cdots (k+1) \,\%\, 131 \Big] \cdot \Big[100(99) \cdots (2k+1) \,\%\, 131 \Big] \cdot \Big[100(99) \cdots (n-3k+1) \,\%\, 131 \Big] \Big] - 1 + 131 \Bigg] \,\%\, 131$$

```
int solucion()
{
    int respuesta = 0, k, n, mod = 131, M = 100;
    int M_fact_pot = potencia_modulo(f_modulo[M], mod-4, mod);
    cin>>n>>k;

    //Resolviendo la expresión por partes.
    respuesta = M_fact_pot * f_modulo[n] % mod;
    respuesta = respuesta * prod_reverso[k] % mod;
    respuesta = respuesta * prod_reverso[2 * k] % mod;
    respuesta = respuesta * prod_reverso[n - 3 * k] % mod;

    //Sumamos mod para evitar resultados negativos.
    respuesta = (respuesta - 1 + mod) % mod;

    return respuesta;
}
```

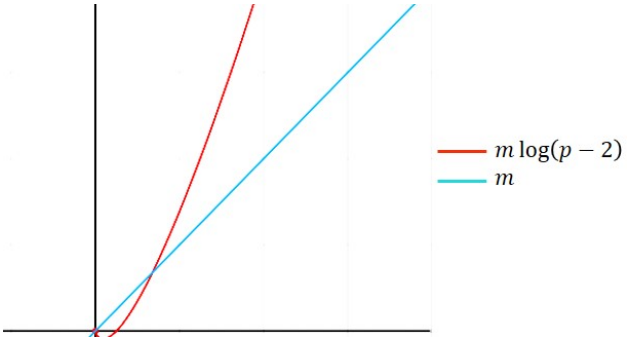
Si lo deseado es resolver una división que involucra un factorial, o un producto de factoriales, como denominador, ¿es preferible usar el *Método 1* o el *Método 2 (factorial reverso)*?

La complejidad del *Método 1* depende de la obtención de la potencia a^{p-2}, mis-

ma que no puede ser preprocesada si a es parte de la entrada del algoritmo solución. Con la técnica de *exponenciación rápida*, que veremos más adelante, podremos realizar dicha potencia de una manera logarítmica. De aquí, la complejidad del *Método 1* es $O(log(p-2))$.

Del otro lado, tenemos el método del factorial reverso, que requiere el trabajo extra de preprocesar tres procedimientos: el producto desde M hasta las k_i's, el factorial de M y la potencia de $M!$, pero que brinda la propiedad de que cada consulta se haga de manera *constante*, esto es $O(1)$.

Ahora, si para cada uno de estos algoritmos se realizan m consultas, el *Método 1* tendrá una complejidad de $O(m \cdot log(p-2))$, y el *Método 2* de $O(m)$. Como lo muestra la gráfica de abajo, para valores pequeños de m será más eficiente utilizar el método 1, sin embargo, cuando se requiera trabajar con grandes datos, será mejor aplicar factorial reverso, pese al trabajo extra inicial que éste implica.



2.3.2. Rolling Hash: Una aplicación de la Aritmética Modular

Comencemos pensando en un ejercicio: Dada una cadena $\mathfrak{C}$ de tamaño c de caracteres, y una cadena $\mathfrak{S}$ de s caracteres, para $s \leq c$, hay que determinar si $\mathfrak{S}$ se encuentra dentro de $\mathfrak{C}$.

Sin el conocimiento del tema que abordaremos aquí, es probable que lo primero

en lo que pensaste fue en verificar, dentro de $\mathfrak{C}$, para cada subcadena de tamaño s, si todos los caracteres allí coinciden con los de $\mathfrak{S}$ (en el mismo orden). ¿Cuál es la complejidad de resolver el problema de esta manera?

Para saber cuántas subcadenas de tamaño s existen dentro de $\mathfrak{C}$, hay que ver que cada elemento en $\mathfrak{C}$ es el primer caracter de alguna subcadena de tamaño s, a excepción de los $s-1$ últimos, que no logran "completar" una subcadena. Así, contamos $c-(s-1)$ subcadenas de tamaño s. Luego, la revisión de cada una de esas subcadenas requiere de s validaciones. Deducimos entonces que la complejidad de este procedimiento es $O(s\cdot(c-s+1))$. Podemos reducir la expresión obtenida si tomamos en cuenta que s siempre será menor o igual que c, pudiendo conservar únicamente los términos dominantes: $O(s\cdot c)$.

Aunque la solución que acabamos de discutir funciona, en el caso de que c fuera 10^5 y s 10^4, por ejemplo, la complejidad del algoritmo llegaría a 10^9, número de operaciones que ya no pasan en tiempo, y habrá que buscar otra forma de resolver el problema.

En un proceso de *Hash*, como se detalla en [15], comenzamos por asignar a cada caracter en juego un entero no negativo (lo que puedes establecer de acuerdo a tus necesidades). En este caso, utilizaremos únicamente letras mayúsculas del alfabeto latino moderno: A=0, B=1, C=2, ..., Y=24, Z=25. Pensemos en el alfabeto como los dígitos de un sistema de numeración en base 26. Ya que nuestro sistema típico es decimal, requerimos trabajar con el equivalente en base 10 de las cadenas; por ejemplo, la palabra "EJEMPLO" sería equivalente al desarrollo:

$$E(26)^6+J(26)^5+E(26)^4+M(26)^3+P(26)^2+L(26)^1+O(26)^0$$

$$=4(26)^6+9(26)^5+4(26)^4+12(26)^3+15(26)^2+11(26)^1+14(26)^0$$

$$=1,344,644,744$$

De esta forma, comparar a $\mathfrak{S}$ con cada subcadena de $\mathfrak{C}$ pasa de revisar cada par de caracteres a una única comparación numérica, que se realiza con complejidad *constante*.

Nota que para una palabra de siete letras obtuvimos un número mayor a 10^9 (en

base 10), lo que sería un problema si estuviéramos tratando con arreglos de gran tamaño, y es justo aquí donde entran las propiedades de aritmética modular, pero surge otra pregunta: ¿cuál sería el módulo más apropiado?

El problema de usar la representación modular de las cadenas es que puede haber más de una cuyo hash sea el mismo; estas situaciones se conocen como *colisiones*. La primera acción que tomaremos para evitarlo será utilizar un módulo muy grande, que suele ser la raíz cuadrada del número más alto con el que se puede trabajar para evitar un desbordamiento en caso de que se presente un producto, digamos 10^9; sin embargo, por grande que sea el número, no podemos garantizar que será suficiente para evitar colisiones. Una buena idea para reducir más las probabilidades de colisiones es usar pares de hashes en lugar de únicamente uno, y el segundo hash se puede obtener utilizando una base distinta (de tamaño mayor o igual que el alfabeto usado para evitar que dos letras se relacionen al mismo número), un módulo distinto, o ambos, pero eso sí, muy cercanos a los del primer hash.

Si obtenemos las potencias a partir de un preproceso, obtener el hash de una subcadena tendrá una complejidad de $O(s)$; la comparación de $\mathfrak{S}$ con cada subarreglo en $\mathfrak{C}$ será $O(1)$, y dado que siguen siendo $c - s + 1$ subcadenas que comparar, la complejidad final será de $O((c - s + 1)(s + 1)) \approx O(c \cdot s)$, que como verás, no es mejor que la solución que propusimos al principio, y es aquí donde entra el proceso de *Rolling Hash*.

Llamemos $\mathcal{C}_i$ a la subcadena de $\mathfrak{C}$ de tamaño s que comienza con $\mathfrak{C}[i]$. Piensa, por ejemplo, en cada par de subarreglos $\mathcal{C}_i$, $\mathcal{C}_{i+1}$; nota que obtener el hash de ambos implica redundancia en el trabajo, pues los únicos caracteres en los que difieren son el primero y el último, lo que nos lleva a pensar que podemos encontrar el hash de $\mathcal{C}_{i+1}$ si conocemos el de $\mathcal{C}_i$.

Volvamos a la palabra "EJEMPLO", para la que consideraremos las subcadenas de tamaño 4, y obtengamos el hash de $\mathcal{C}_0$ ("EJEM") y de $\mathcal{C}_1$ ("JEMP"):

$$h(\mathcal{C}_0) = \left[4(26)^3 + 9(26)^2 + 4(26)^1 + 12(26)^0\right] \% \, 10^9 = 76,504$$

$$h(\mathcal{C}_1) = \left[9(26)^3 + 4(26)^2 + 12(26)^1 + 15(26)^0\right] \% \, 10^9 = 161,215$$

Pero $\mathcal{C}_1$ también se obtiene de la siguiente manera:

$$h(\mathcal{C}_1) = \left[\left[\left(4(26)^3 + 9(26)^2 + 4(26)^1 + 12(26)^0\right) - \left(4(26)^3\right)\right] \cdot 26 + 15\right] \% \ 10^9$$

$$= \left[\left[h(\mathcal{C}_0) - (\mathfrak{C}[0])(26)^3\right] \cdot 26 + h(\mathfrak{C}[4])\right] \% \ 10^9$$

Esta misma técnica puede ser usada para el resto de las subcadenas, que compartirán siempre $s-1$ caracteres con el subarreglo anterior.

Con este método, el hash de todas las subcadenas de $\mathfrak{C}$ se obtiene con complejidad $O(c)$, y dado que la comparación con $\mathfrak{S}$ es constante, el tiempo en que se ejecuta todo el algoritmo es $O(c)$.

Formalizando, para un número natural m y una base B, el rolling hash de la subcadena $\mathcal{C}_0$, de tamaño s, obedece al siguiente procedimiento:

$$h(\mathcal{C}_0) = \left[\sum_{i=0}^{s-1} \mathfrak{C}[i] \cdot B^{s-1-i}\right] \% \ m$$

Y el rolling hash de cada subcadena $\mathcal{C}_i$, para $1 \leq i \leq c-s+1$, se obtiene de la ecuación recursiva:

$$h(\mathcal{C}_i) = \left[\left(h(\mathcal{C}_{i-1}) - \mathfrak{C}[i-1] \cdot B^{s-1}\right) \cdot B + \mathfrak{C}[i+s-1]\right] \% \ m$$

Nuevamente, recordemos que para evitar colisiones, es recomendable que se manejen dos versiones de Hash de cada cadena.

Ejemplo 75. Imagina una sopa de letras en la que únicamente se pueden hallar palabras en dirección horizontal de izquierda a derecha, pero que admite encontrar secciones de palabras siempre que éstas se hayan seccionado en dos partes y la primera y la última parte se encuentre al final de un renglón y al inicio del renglón

siguiente, respectivamente.

Considera como entrada del problema las dimensiones de la sopa b y h, seguidas de h líneas con exactamente b letras mayúsculas cada una; luego vendrán dos enteros positivos w y t: w será el número de palabras a buscar y t el tamaño de ellas; finalmente tendrás w líneas, todas con una palabra distinta.

El reto será devolver la posición del primer y el último elemento de cada palabra en el orden en que fueron dadas, en caso de que esté presente en la sopa de letras; de lo contrario, simplemente imprimiremos "No encontrada". Consideraremos al primer elemento de la sopa de letras como la posición $(1, 1)$. Podemos asegurar que cada palabra aparece a lo sumo una vez.

> **Solución.** Por la naturaleza de la sopa de letras, en lugar de buscar una palabra en cada renglón, podemos concatenar todas las líneas como una sola, obteniendo una cadena de $b \cdot h$ letras, a la que aplicaremos dos procesos de rolling hash con distintas bases y distintos módulos.
>
> Luego, para cada palabra a buscar, aplicaremos dos procesos de rolling hash también, con las mismas condiciones que siguieron los de la sopa de letras, y estos pares serán comparados contra los obtenidos en la cadena de $b \cdot h$. Obviamente, si llegamos a la comparación de la última subcadena en la sopa de letras y no hay coincidencia, se concluye que dicha palabra no se encuentra en el juego.
>
> Finalmente, hay que ocuparnos de devolver la ubicación de la primera y la última letra de cada palabra encontrada: Llamemos (x_1, x_2) a la ubicación de alguno de los caracteres de nuestro interés. Entonces, $x_1 = Y \,\%\, b$, en donde Y es la posición de la letra dentro de la cadena que representa a la sopa de letras, comenzando en 1, esto, con la precaución de que, si el residuo es 0, x_1 se encuentra al final de un renglón, es decir, $x_1 = b$; por otro lado, la expresión que nos proporciona el valor de x_2 es $x_2 = ((Y - x_1)/b) + 1 = \lceil Y/b \rceil$, lo que describimos como el renglón en el que se encuentra nuestra letra de interés.
>
> Para el rolling hash, utilizaremos los valores $B_1 = 29$, $p_1 = 10^9 + 9$, y

$B_2 = 31$, $p_2 = 10^9 + 7$.

```
map<pair<int, int>, int> hashes;
int B1 = 29, B2 = 31, mod1 = 1000000009, mod2 = 1000000007;

//Obtener hash de cadenas tamaño t.
int obtener_hash(string &cadena, int t, int B, int mod)
{
    int i;
    long long resp = 0;
    for (i = 0; i < t; i = i + 1)
        resp = (resp * B % mod + cadena[i] - 'a') % mod;
    return resp;
}

int rolling_hash(string &cadena, long long hash, int B, int mod, int t)
{
    int base_pot;
    base_pot = potencia_modulo(B, t - 1, mod);

    //Obteniendo hash por partes.
    hash = (hash - (cadena[i - 1] - 'a') * base_pot % mod + mod) % mod;
    hash = ((hash * B % mod) + cadena[i + t - 1] - 'a') % mod;

    return hash;
}

void preproceso(string &sopa, int t)
{
    int i;

    //Hash de la primera subcadena tamaño t.
    long long hash1 = obtener_hash(sopa, t, B1, mod1);
    long long hash2 = obtener_hash(sopa, t, B2, mod2);
    hashes[{hash1, hash2}] = 1;

    //El resto de los hash.
    for (i = 1; i < sopa.size() - (t - 1); i = i + 1)
    {
        hash1 = rolling_hash(sopa, hash1, B1, mod1, t, i);
        hash2 = rolling_hash(sopa, hash2, B2, mod2, t, i);

        //i+1 porque empezamos en (1,1).
        hashes[{hash1, hash2}] = i + 1;
    }
```

```
44  }
45
46  void solucion()
47  {
48      int b, h, w, t, i, hash1, hash2, Y;
49      string sopa, palabra;
50      cin>>b>>h;
51      cin>>sopa;
52      cin>>w>>t;
53
54      preproceso(sopa, t);
55
56      for (i = 0; i < w; i = i + 1)
57      {
58          cin>>palabra;
59          hash1 = obtener_hash(palabra, t, B1, mod1);
60          hash2 = obtener_hash(palabra, t, B2, mod2);
61
62          //Buscamos la palabra dentro del mapa y la asignamos a Y.
63          Y = hashes[{hash1, hash2}];
64          if ( Y )
65          {
66              cout<<"Palabra "<<i<<":\n";
67              int a = Y % b;
68              if (a == 0) a = b;
69              cout<<"Inicio: "<<a<<" , "<<ceil( Y/b )<<"\n";
70              a = (Y + t - 1) % b;
71              if (a == 0) a = b;
72              cout<<"Fin: "<<a<<" , ";
73              cout<<ceil((Y + t - 1)/b)<<"\n";
74          }
75          else
76          {
77              cout<<"Palabra "<<i<<":\n";
78              cout<<"No encontrada\n";
79          }
80      }
81  }
```

Ejemplo 76. Seguiremos con el ejercicio de la singular sopa de letras, pero en esta ocasión, podremos encontrar palabras en dirección horizontal y vertical.

Solución. El ejemplo anterior nos dice cómo encontrar las palabras de manera horizontal guardando la sopa de letras como una única cadena.

¿Cómo lidiamos con las verticales? Si logramos crear una cadena que resulta de la concatenación de todas las columnas de la sopa de letras, podremos aplicar el mismo proceso que para el problema pasado.

Llamemos H a la cadena obtenida de concatenar las filas y V a la que resulta de concatenar las columnas. Para construir V, nota que cada b caracteres se agregará el siguiente elemento de determinada columna. Así, las primeras h letras de V serán:

$$\begin{aligned} V[1] &= H[1] \\ V[2] &= H[1+b] \\ V[3] &= H[1+2b] \\ &\vdots \\ V[h] &= H[1+(h-1)b] \end{aligned}$$

Luego vendrán las letras:

$$\begin{aligned} V[h+1] &= H[2] \\ V[h+2] &= H[2+b] \\ V[h+3] &= H[2+2b] \\ &\vdots \\ V[2h] &= H[2+(h-1)b] \end{aligned}$$

Generalizando,

$$\begin{aligned} V[ih+1] &= H[i+1] \\ V[ih+2] &= H[(i+1)+b] \\ V[ih+3] &= H[(i+1)+2b] \\ &\vdots \\ V[(i+1)h] &= H[(i+1)+(h-1)b] \end{aligned}$$

En donde $i = 0, 1, 2, 3, \ldots, b-1$.

Posterior a la construcción de V, le aplicaremos rolling hash para comparar con las palabras que no fueron encontradas en dirección horizontal.

Finalmente, hay que obtener la ubicación de la primera y la última letra de las palabras encontradas en V: Para cada lugar (y_1, y_2), $y_1 = \lceil Y/h \rceil$ y $y_2 = Y \,\%\, h$, en donde Y es la ubicación de la letra en cuestión dentro de V (comenzando en 1); recordemos que si $Y \,\%\, h = 0$, debemos cambiar el dato por h.

Reutilizaremos el programa solución del ejercicio anterior y agregaremos lo necesario para resolver este problema.

```
1  void preproceso(string &sopa, int t, bool dir)
2  //dir = true si es horizontal, dir = false si es vertical.
3  {
4      int i;
5
6      //Hash de la primera subcadena tamaño t.
7      long long hash1 = obtener_hash(sopa, t, B1, mod1);
8      long long hash2 = obtener_hash(sopa, t, B2, mod2);
9      hashes[{hash1, hash2}] = {1, dir};
10
11     //El resto de los hash.
12     for (i = 1; i < sopa.size() - (t - 1); i = i + 1)
13     {
14         hash1 = rolling_hash(sopa, hash1, B1, mod1, t, i);
15         hash2 = rolling_hash(sopa, hash2, B2, mod2, t, i);
16
17         //i+1 porque empezamos en (1,1).
18         hashes[{hash1, hash2}] = {i + 1, dir};
19     }
20 }
21
22 void solucion()
23 {
24     int b, h, w, t, i, j;
25     string palabra, H, V;
26     cin>>b>>h;
27     cin>>H;
28     cin>>w>>t;
29
30     for(i = 0; i < b; i = i + 1)
```

```
31          for (j = 1; j < h; j = j + 1)
32              V[(i * h) + j] = H[i + (j * b)];
33
34      preproceso(H, t, true);
35      preproceso(V, t, false);
36
37      for (i = 0; i < w; i = i + 1)
38      {
39          cin>>palabra;
40          int hash1 = obtener_hash(palabra, t, B1, mod1);
41          int hash2 = obtener_hash(palabra, t, B2, mod2);
42          int Y = hashes[{hash1, hash2}].first;
43          bool dir = hashes[{hash1, hash2}].second;
44
45          if ( Y )
46          {
47              if ( dir )
48              {
49                  cout<<"Palabra "<<i<<":\n";
50                  int a = Y % b;
51                  if (a == 0) a = b;
52                  cout<<"Inicio: "<<a<<" , "<<ceil( Y/b )<<"\n";
53                  a = (Y + t - 1) % b;
54                  if (a == 0) a = b;
55                  cout<<"Fin: "<<a<<" , ";
56                  cout<<ceil((Y + t - 1)/b)<<"\n";
57              else
58              {
59                  cout<<"Palabra "<<i<<":\n";
60                  int a = Y % h;
61                  if (a == 0) a = h;
62                  cout<<"Inicio: "<<ceil( Y/h )<<" , "<<Y % h<<"\n";
63                  cout<<"Fin: "<<ceil((Y + t - 1)/h)<<" , ";
64                  a = (Y + t - 1) % h;
65                  if (a == 0) a = h;
66                  cout<<(Y + t - 1) % h<<"\n";
67              }
68          }
69          else
70          {
71              cout<<"Palabra "<<i<<":\n";
72              cout<<"No encontrada.\n";
73          }
74      }
75  }
```

2.4. Criterios de Divisibilidad

Es común encontrarnos con números extremadamente grandes cuyo tratamiento se vuelve complicado debido a factores como memoria, tipos de datos y eficiencia. Cuando se habla de la multiplicidad de un número, es decir, de probar que éste es divisible por algún otro, un buen tratamiento para el problema es el uso de criterios de divisibilidad, mismos que estudiaremos en esta sección.

Es claro que todo entero es múltiplo de 1 y que, un número cuyo último dígito es par es múltiplo de 2, por lo tanto, comenzaremos a discutir los criterios de divisibilidad desde el número 3.

Proposición 16. Un número entero x es múltiplo de 3 si la suma de los dígitos de x es múltiplo de 3.

Demostración. Sea $x_n x_{n-1} \dots x_2 x_1 x_0$ la representación ordenada de los dígitos de x. Entonces:

$$\begin{aligned} x &\equiv x \ (mod\ 3) \\ &\equiv x_n x_{n-1} \dots x_2 x_1 x_0 \ (mod\ 3) \\ &\equiv \left(x_n \cdot 10^n + x_{n-1} \cdot 10^{n-1} + \cdots + x_1 \cdot 10 + x_0 \right) \ (mod\ 3) \end{aligned}$$

Como $10 \equiv 1 \ (mod\ 3)$, se cumple que $10^k \equiv 1^k \equiv 1 \ (mod\ 3)$ para todo entero k. Por lo tanto:

$$\begin{aligned} x &\equiv \left(x_n \cdot 1 + x_{n-1} \cdot 1 + \cdots + x_1 \cdot 1 + x_0 \right) \ (mod\ 3) \\ &\equiv \left(x_n + x_{n-1} + \cdots + x_1 + x_0 \right) \ (mod\ 3) \end{aligned}$$

Luego, si $x_n + x_{n-1} + \cdots + x_1 + x_0$ es congruente con 0 módulo 3, x también lo será, lo que, a su vez, nos dice que x sera divisible por 3.

□

Nota que si la suma de los dígitos de x resulta en algo mayor a 10, es posible sumar también los dígitos de ese número y repetir ese proceso tantas veces como desees hasta que obtengas un número lo suficientemente pequeño como para que tengas la

certeza de que éste es múltiplo de 3, pues lo que hemos probado arriba es que todo entero es congruente con la suma de sus dígitos, módulo 3.

Proposición 17. Un número entero x es múltiplo de 4 si el número formado por los últimos 2 dígitos de x es múltiplo de 4, o bien, si ambos son 0's.

Demostración. Sea $x_n x_{n-1} \dots x_2 x_1 x_0$ la representación digital de x. Entonces, x se puede expresar como $x = x_n \cdot 10^n + x_{n-1} \cdot 10^{n-1} + \cdots + x_1 \cdot 10 + x_0$. A su vez, lo anterior se puede escribir como:

$$x = \Big(x_n x_{n-1} \dots x_2\Big) \cdot 10^2 + \Big(x_1 x_0\Big)$$

Es claro que $4 \Big| (x_n x_{n-1} \dots x_2) \cdot 10^2$ ya que $4 \Big| 10^2$. Por tanto, $4 \Big| x$ si el número $x_1 x_0$ es múltiplo de 4 también, o bien, si x_1 y x_0 no suman nada, o sea, sin ambos son 0.

□

Proposición 18. Un entero x es divisible por 5 si su último dígito es 0 o 5.

Demostración. Sea $x_n x_{n-1} \dots x_2 x_1 x_0$ la representación en dígitos de x, mismo que también puede ser escrito como $x = (x_n x_{n-1} \dots x_2 x_1) \cdot 10 + x_0$.

Es claro que $5 \Big| 10$, por lo que $5 \Big| (x_n x_{n-1} \dots x_2 x_1) \cdot 10$. Entonces que x sea múltiplo de 5 depende de que x_0 lo sea.

□

Proposición 19. Sea x un entero cuya representación digital es $x_n x_{n-1} \dots x_2 x_1 x_0$. x es múltiplo de 7 si y sólo si la diferencia entre $x_n x_{n-1} \dots x_2 x_1$ y $2x_0$ también lo es.

Demostración. Por comodidad, llamaremos m a $(x_n x_{n-1} \dots x_2 x_1)$, con lo que $x = 10m + x_0$.

Probaremos primero que si $m - 2x_0$ es múltiplo de 7, entonces x también lo es: Existe algún entero k_1 para el que $m - 2x_0 = 7k_1$. Si multiplicamos por 10 y luego sumamos x_0 a ambos lados:

$$\begin{aligned} m - 2x_0 &= 7k_1 \\ 10m - 20x_0 &= 70k_1 \\ 10m - 20x_0 + x_0 &= 70k_1 + x_0 \end{aligned}$$

$$\begin{aligned} &\implies 10m + x_0 = 70k_1 + 21x_0 \\ &\implies x = 7(10k_1 + 3x_0) \end{aligned}$$

De aquí, deducimos que 7 divide a x y terminamos la primera parte de la demostración.

Para completar la demostración, supongamos que se cumple que $7 \Big| x$ para demostrar que $7 \Big| m - 2x_0$: Podemos asegurar la existencia de un entero k_2 que cumple que $x = 10 \cdot m + x_0 = 7k_2$. Sustrayendo $21x_0$ de ambas partes:

$$\begin{aligned} 10m + x_0 &= 7k_2 \\ 10m + x_0 - 21x_0 &= 7k_2 - 21x_0 \\ \implies 10m - 20x_0 &= 7k_2 - 21x_0 \\ \implies 10(m - 2x_0) &= 7(k_2 - 3x_0) \end{aligned}$$

Como $k_2 - 3x_0 \in \mathbb{Z}$, $7 \Big| 10(m - 2x_0)$, pero es claro que $7 \not| 10$, por lo que forzosamente tiene que ocurrir que $7 \Big| m - 2x_0$.

□

Computacionalmente hablando, este método tiene un problema, y es que el número no se puede tratar como una cadena, pues requerimos aplicar una resta. Esto dificulta mucho su implementación, por lo que parece una mejor idea aplicar un procedimiento de *hash* para comprobar la divisibilidad por 7 de un número de más de 18 dígitos.

Proposición 20. Un número entero x es múltiplo de 8 si el número formado por los últimos 3 dígitos de x es múltiplo de 8, o bien, si todos ellos son 0's.

Demostración. Este criterio se prueba de manera muy similar al de 4.

Sea $x_n x_{n-1} \dots x_2 x_1 x_0$ la representación ordenada de los dígitos de x. Entonces, $x = \Big(x_n x_{n-1} \dots x_3\Big) \cdot 10^3 + \Big(x_2 x_1 x_0\Big)$. Del hecho de que 8 divide a $1,000$, podemos asegurar que también divide a $\Big(x_n x_{n-1} \dots x_3\Big) \cdot 10^3$, por lo que lo único que se requiere para que x sea múltiplo de 8 es que el número $x_2 x_1 x_0$ también lo sea, o que

no contribuya con nada a x, siendo los tres dígitos 0.

□

Proposición 21. Un número entero x es múltiplo de 9 si la suma de los dígitos de x es múltiplo de 9.

Demostración. Probaremos que todo entero x es congruente con la suma de sus dígitos módulo 9 (como en el criterio de 3). De esa forma, si la suma de los dígitos es múltiplo de 9, x también lo será.

Nuevamente, necesitamos la representación digital ordenada de x: $x_n x_{n-1} \dots x_2 x_1 x_0$, de lo que se sigue:

$$\begin{aligned} x &\equiv x_n x_{n-1} \dots x_2 x_1 x_0 \ (mod\ 9) \\ &\equiv x_n(10)^n + x_{n-1}(10)^{n-1} + \dots + x_2(10)^2 + x_1(10) + x_0 \ (mod\ 9) \\ &\equiv x_n(1)^n + x_{n-1}(1)^{n-1} + \dots + x_2(1)^2 + x_1(1) + x_0 \ (mod\ 9) \\ &\equiv x_n + x_{n-1} + \dots + x_2 + x_1 + x_0 \ (mod\ 9) \end{aligned}$$

Con lo que concluimos que, efectivamente, cualquier entero es congruente con la suma de sus dígitos, módulo 9.

□

Proposición 22. Sea S_{par} la suma de los dígitos en las posiciones pares y sea S_{impar} la suma de los dígitos en las posiciones impares de un un número entero x. x es múltiplo de 11 si $|S_{par} - S_{impar}|$ es también un múltiplo de 11.

Demostración. Establezcamos $x = x_n x_{n-1} \dots x_2 x_1 x_0 = \sum_{i=0}^{n} x_i(10)^i$.

Ya que $10 \equiv -1\ (mod\ 11)$, es fácil ver que, para un entero no negativo k, $10^k \equiv (-1)^k \equiv 1\ (mod\ 11)$ si k es par, y $10^k \equiv (-1)^k \equiv -1\ (mod\ 11)$ si k es impar.

Considerando que $x = x_n \cdot 10^n + x_{n-1} \cdot 10^{n-1} + \dots + x_1 \cdot 10 + x_0$, sabemos que todos los elementos de S_{par} son congruentes con $1 \cdot x_i$ módulo 11, para $0 \leq i \leq n$, $i = par$, y que todos los elementos de S_{impar} son congruentes con $-1 \cdot x_i$ módulo 11 para $0 \leq i \leq n$, $i = impar$. Por lo tanto,

$$\begin{aligned} x &\equiv x_n x_{n-1} \cdots x_1 x_0 \ (mod\ 11) \\ &\equiv \left(S_{par} - S_{impar}\right) \ (mod\ 11) \end{aligned}$$

Así, x será múltiplo de 11 si $S_{par} - S_{impar} \equiv 0 \ (mod\ 11)$, es decir, si $11 \Big| (S_{par} - S_{impar})$. □

Los anteriores son los criterios de divisibilidad más utilizados, sin embargo, es posible deducir algunos otros a partir de los anteriores. El criterio de divisibilidad para un número que no es primo ni una potencia de un primo es el conjunto de los criterios de divisibilidad para los números resultantes de su descomposición canónica. Por ejemplo, para probar que un número es múltiplo de 6, bastará con revisar que éste sea múltiplo de 2 y de 3 (que su último dígito sea par y que la suma de sus dígitos sea múltiplo de 3).

Ejemplo 77. Dado un entero n de más de 18 dígitos, determina si éste es múltiplo de 18.

Solución. Debido al tamaño de la entrada, no será posible manejarla como un entero, así que tendrá que ser leída como una cadena de caracteres.

Podemos factorizar a 18 como $2 \cdot 9$, y será ésta la descomposición de la que haremos uso para probar que n es divisible entre 18. Así, requeriremos que $n[longitud - 1]$ (si el primer elemento es $n[0]$) sea múltiplo de 2, y que $\sum_{i=0}^{longitud-1} n[i]$ sea múltiplo de 9.

```
1  bool solucion()
2  {
3      string numero;
4      int suma_digitos = 0, ultimo_digito;
5      cin>>numero;
6
7      //Para 2.
8      ultimo_digito = numero[numero.size() - 1] - '0';
9
10     //Para 9.
11     for (char n : numero)
12         suma_digitos = suma_digitos + n - '0';
13
14     if (ultimo_digito % 2 == 0 and suma_digitos % 9 == 0)
15         return true;
16     else
```

```
17          return false;
18  }
```

Ejemplo 78. Dados dos enteros positivos m y n mayores que 10^{20}, determina si $m+n$ es un múltiplo de 4.

> **Solución.** No importa cuán grande sean m y n, únicamente debemos enfocar nuestra atención en los dos últimos caracteres de ambos, por lo que convertiremos a enteros las dos últimas posiciones de cada uno y los sumaremos. Si éste es múltiplo de 4, la suma de $m+n$ también lo es.
>
> Nota que existe la posibilidad de que $m+n$ sea mayor que 99, pero aun en esos casos, nuestra solución no se modificará, pues comprobar que un número de tres dígitos es múltiplo de 4 es equivalente a comprobar que $100k$, $k \in \mathbb{Z}$ sumado al número de los dos últimos dígitos lo es (y como vimos en la demostración del criterio de divisibilidad de 4, $4 \mid 100$).

```
1   bool solucion()
2   {
3       string m, n;
4       int m0, m1, n0, n1, para_m, para_n;
5       cin>>m>>n;
6
7       //Obtenemos los números formados por los dos últimos dígitos.
8       m0 = m[m.size() - 1] - '0';
9       m1 = m[m.size() - 2] - '0';
10      n0 = n[n.size() - 1] - '0';
11      n1 = n[n.size() - 2] - '0';
12      para_m = 10 * m1 + m0;
13      para_n = 10 * n1 + n0;
14
15      if (para_m + para_n % 4 == 0)
16          return true;
17      else
18          return false;
19  }
```

Ejemplo 79. El gobierno de cierto territorio cuya población asciende a exactamente 440 adultos y un número indefinido de infantes intenta impulsar la agricultura. Para

ello, ha decidido repartir la superficie de terrenos con los que cuentan, con la única condición de que éstos sean trabajados apropiadamente; no obstante, antes de asignar las tierras a los nuevos dueños, el gobierno se quiere asegurar de que las hectáreas puedan ser repartidas de manera equitativa entre todos los pobladores adultos de tal forma que todos reciban hectáreas completas.

Hay que construir un programa que determine si es posible proceder con la repartición de terrenos o habrá que hacer una compra de tierras extra. Para esto, considera que el pueblo es altamente afortunado en recursos naturales y la mínima cantidad de hectáreas que podrían estar disponibles es 10^{19}.

> **Solución.** $440 = 5 \cdot 8 \cdot 11$, por lo tanto habrá que validar que la superficie de terreno de la que se dispone sea múltiplo de 5, de 8 y de 11, y lo haremos haciendo uso de los criterios de divisibilidad correspondientes. Para ello, denotaremos como H al número de hectáreas.

```
1  bool solucion()
2  {
3      string H;
4      int ultimo_digito, penultimo, antepenultimo;
5      int para_8, S_par = 0, S_impar = 0, i;
6      cin>>H;
7
8      //Para 5.
9      ultimo_digito = H[H.size() - 1] - '0';
10
11     //Para 8.
12     penultimo = H[H.size() - 2] - '0';
13     antepenultimo = H[H.size() - 3] - '0';
14     para_8 = 100 * antepenultimo + 10 * penultimo + ultimo_digito;
15
16     //Para 11.
17     for (i = 0; i < H.size(); i = i + 1)
18     {
19         if (i % 2 == 0)
20             S_par = S_par + H[i];
21         else
22             S_impar = S_impar + H[i];
23     }
24
25     if ( (ultimo_digito % 5 == 0) and (para_8 % 0 == 0) and (S_par -
   ↪  S_impar % 11 == 0) )
```

```
26          return true;
27      else
28          return false;
29  }
```

Ejemplo 80. Considera un entero x en base B, ambos datos dados. ¿Cuál es la raíz digital[3] de x (en base B)?

Solución. En la demostración del criterio de divisibilidad de 9 vimos que todo entero es congruente con la suma de sus dígitos módulo 9, y en el problema 64 pudimos observar que la raíz digital de un número es el mismo que el resultado de aplicarle la operación módulo 9, con la reserva del 0, que debe ser intercambiado por 9. Entonces comencemos por analizar si sucede lo mismo en cualquier base.

Ya que x es dado en base B, podemos expresar a x de la siguiente forma:

$$x = x_n B^n + x_{n-1} B^{n-1} + \cdots + x_2 B^2 + x_1 B^1 + x_0 B^0$$

En donde $x_n x_{n-1} \ldots x_2 x_1 x_0$ es la representación digital de x.

Además, $B \equiv 1 \ (mod\ B-1)$, por lo que $B^k \equiv 1^k \equiv 1 \ (mod\ B-1)$ para todo entero k. Entonces:

$$\begin{aligned} x &\equiv x_n B^n + x_{n-1} B^{n-1} + \cdots + x_2 B^2 + x_1 B^1 + x_0 B^0 \ (mod\ B-1) \\ &\equiv x_n (1)^n + x_{n-1} (1)^{n-1} + \cdots + x_2 (1)^2 + x_1 (1) + x_0 \ (mod\ B-1) \\ &\equiv x_n + x_{n-1} + \cdots + x_2 + x_1 + x_0 \ (mod\ B-1) \end{aligned}$$

Con esto vemos que cualquier número x en base B es congruente con la suma de sus dígitos módulo $B-1$, con lo que deducimos que x es congruente con su raíz digital módulo $B-1$, y que se sostiene una biyección

[3] La raíz digital de un entero x, como se vio en el problema 64, se obtiene de sumar sus dígitos, luego sumar los dígitos del número resultante, y repetir el proceso cuantas veces sea necesario para obtener únicamente un dígito.

entre el resultado y el conjunto $\{0, 1, 2, \ldots, B-1\}$; nuevamente hay que considerar que si lo obtenido del módulo es 0, deberemos devolver $B-1$.

Visto esto y considerando el hecho de que el álgebra sobre la que desarrollamos las propiedades de aritmética modular es de un sistema de numeración en base 10, tendremos que desarrollar el número dado como $x_n B^n + x_{n-1} B^{n-1} + \cdots + x_2 B^2 + x_1 B^1 + x_0 B^0$, para aplicarle módulo $B-1$, y al resultado de esta operación, devolverlo a la base B, que será un único caracter. (El procedimiento será muy similar a la obtención del hash de una cadena).

```
1  char solucion()
2  {
3      string x;
4      int B, raiz_digital = 0;
5      cin>>x>>B;
6
7      for (char c: x)
8      {
9          if ( isalpha(c) )
10             raiz_digital = (raiz_digital * B % (B-1) + cadena[i] - 'a'
11                         + 10) % (B-1);
12         else
13             raiz_digital = (raiz_digital * B % (B-1) + cadena[i] - '0')
14                         % (B-1);
15     }
16
17     if (raiz_digital == 0)
18     {
19         if (B >= 10)
20             return B + 'a' - 10;
21         else
22             return B + '0';
23     }
24     else if (raiz_digital < 10)
25         return raiz_digital + '0';
26     else
27         return raiz_digital + 'a' - 10;
28 }
```

2.5. Máximo Común Divisor

El máximo común divisor de dos enteros a y b es el mayor de los divisores que a y b comparten, y se denota como (a, b), aunque es común encontrar expresiones como $mcd(a, b)$ y $gcd(a, b)$ por sus siglas en inglés (greatest common divisor). En esta obra en particular, usaremos la notación de $mcd(a, b)$. Ya que el máximo común divisor de cualesquiera par de enteros a y b es el mismo que el de $|a|$ y $|b|$, todos los resultados que se presentarán en esta sección serán sobre enteros positivos, y será aplicable al resto.

Si $mcd(a, b) = 1$, es decir, si a y b no tienen factores mayores que 1 en común, se dice que a y b son *primos relativos* o *coprimos*.

2.5.1. Algoritmo de Euclides

El método más intuitivo para obtener el máximo común divisor de dos números es obtener todos los factores de ambos e identificar el mayor de entre aquéllos que comparten, sin embargo, existe una vía mejor, cuyas bases descansan en los siguientes resultados.

Antes de abordar el siguiente teorema, veamos con un ejemplo lo que éste probará: Piensa en un entero a que deseemos dividir por algún otro entero b, digamos, 38 y 7, respectivamente:

$$\begin{array}{r|l} & 5 \\ \hline 7 & 38 \\ & 3 \end{array}$$

Esta operación tan conocida y tan sencilla nos dice que podemos expresar a 38 como $(7 \times 5) + 3$. Pues bien, el siguiente teorema probará que es posible hallar una descomposición *única* de este tipo para cualquier par de enteros.

Teorema 12. [Algoritmo de la División]. Sean a y b dos números enteros distintos de 0. Existe una pareja única de enteros q y r, tales que $0 \leq r < |b|$, que satisface:

$$a = bq + r$$

(q se conoce como cociente y r como residuo).

Demostración. Sea $q \cdot b$ el mayor múltiplo de b menor o igual que a; llamemos r a la diferencia entre a y $q \cdot b$; podemos asegurar que r es menor que b, pues $q \cdot b \leq a < (q+1)b$, y la diferencia entre $(q+1)b$ y $q \cdot b$ es b. Con esto, probamos que, efectivamente, existe la pareja (q, r).

Para probar la unicidad de dicha pareja, supongamos que existe otra pareja que cumple la condición, digamos q_1, r_1. De aquí, $a = bq + r = bq_1 + r_1 \Longrightarrow bq - bq_1 = r_1 - r \Longrightarrow b(q - q_1) = r_1 - r$. Como q y q_1 son enteros, su diferencia también lo es, lo que implica que $b \Big| r_1 - r$.

Ya que estamos trabajando con la idea de que r y r_1 son distintos, $r_1 - r \neq 0$; además, para que b pueda ser divisor de $r_1 - r$, $b \leq |r_1 - r|$, pero, por construcción, r y r_1 deben ser menores que $|b|$, con más razón la expresión $|r_1 - r|$, lo que implica una contradicción.

Por último, es claro que si $r = 0$, $a = b \cdot q$, o sea que $b \Big| a$.

□

Ya hemos probado que hay una única pareja (q, r) para la que $a = bq + r$, y esa será la base de nuestros siguientes resultados, mismos que nos proporcionarán un método eficiente para encontrar el máximo común divisor de dos enteros.

Teorema 13. Sean a y b dos enteros distintos de 0. El máximo común divisor de a y b es el mismo que el de b y r, donde r es el residuo de la división $a \div b$.

Demostración. Como ya vimos, podemos expresar a a de la siguiente forma: $a = bq + r$, $q, r \in \mathbb{Z}$.

Consideremos a algún divisor común d de a y b. Para que esto sea cierto y de acuerdo a la ecuación mencionada en el párrafo anterior, debe ocurrir que $d \Big| r$ (porque $d \Big| a$ y

$d \mid b$).

Luego, pensemos en algún divisor común d_1 de b y r. De la igualdad $a = bq + r$, podemos deducir que $d_1 \mid a$. Entonces cualquier divisor común de b y r también lo será de a.

Por tanto, todos los divisores comunes de a y b serán los mismos que los de b y r, en particular, lo será el mayor de ellos.

□

Teorema 14. [Algoritmo de Euclides]. Nuevamente, considera dos enteros a y b para los que se cumple que $a = bq + r$ para $q, r \in \mathbb{Z}$. Llamemos r_0 a a y r_1 a b; si aplicamos repetidamente el algoritmo de la división, se obtendrán las siguientes relaciones:

$$\begin{aligned}
r_0 &= r_1 q_1 + r_2, && 0 < r_2 < r_1 \\
r_1 &= r_2 q_2 + r_3, && 0 < r_3 < r_2 \\
&\vdots \\
r_{n-2} &= r_{n-1} q_{n-1} + r_n, && 0 < r_n < r_{n-1} \\
r_{n-1} &= r_n q_n + r_{n+1}, && r_{n+1} = 0
\end{aligned}$$

Y el máximo común divisor de a y b será r_n: el último residuo distinto de 0.

Demostración. Dado que la sucesión $r_0, r_1, \ldots, r_n, r_{n+1}$ es decreciente y todos los r_i son no negativos, habrá una iteración en la que el residuo será nulo (cuando $r_{n+1} = 0$, lo que implicará que $r_n \mid r_{n-1}$).

En el teorema anterior demostramos que, para un par de enteros a, b para los que $a = bq + r$, se cumpliría que $mcd(a, b) = mcd(b, r)$, por lo que es cierto que $mcd(a, b) = mcd(r_0, r_1) = mcd(r_1, r_2) = mcd(r_2, r_3) = \cdots = mcd(r_n, r_{n+1})$, pero ya dijimos que $r_n \mid r_{n-1}$, por lo que el máximo común divisor de r_n y r_{n+1} (=0) será r_n.

Finalmente, por transitividad, $mcd(r_n, r_{n+1}) = mcd(a, b) = r_n$.

□

2.5 Máximo Común Divisor

Llevemos el teorema anterior a un escenario más práctico buscando el máximo común divisor de $a = 60$ y $b = 36$.

a	b	$r = a \,\%\, b$
60	36	24
36	24	**12**
24	12	0

La primera división en la que tenemos que pensar es $60 \div 36$, que arroja como residuo a 24; luego, a adopta el valor de 36 y b el de 24 (lo que antes era b ahora es a, y lo que antes era r ahora es b), dando como residuo 12, y éste será justamente el máximo común divisor de ambos números, pues es la última iteración en la que el residuo será distinto de 0, lo que es lo mismo, el valor de b en la iteración en la que $r = 0$.

El siguiente ejemplo nos muestra que los números 39 y 11 son primos relativos, pues su máximo común divisor es 1.

a	b	$r = a \,\%\, b$
39	11	6
11	6	5
6	5	**1**
5	1	0

Ejemplo 81. Encuentra el máximo común divisor de dos enteros dados a y b.

Solución. Aplicando el algoritmo de Euclides:

```
1  int A_Euclides()
2  {
3      int a, b, r;
4      cin>>a>>b;
5
6      r = a;
7      do
8      {
9          a = b;
10         b = r;
11         r = a % b;
```

```
12      } while (r != 0)
13
14      return b;
15  }
```

En la ejecución del algoritmo de Euclides, el peor de los casos ocurre cuando tenemos dos números de Fibonacci consecutivos F_n y F_{n-1}, según [16]. Así, dado que $F_n = F_{n-1} + F_{n-2}$, al aplicar el algoritmo tendremos:

$$mcd(F_n, F_{n-1}) = mcd(F_{n-1}, F_{n-2}) = mcd(F_{n-2}, F_{n-3}) = \cdots = mcd(1, 0) = 1$$

Fueron necesarias n iteraciones para obtener el mcd, y dado que $F_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$, se sigue que $n \approx log(F_n) = log(F_{n-1} + F_{n-2})$. Así, en el peor de los casos, para dos enteros a y b, la complejidad del algoritmo de Euclides será $O(log(a + b))$.

Como vimos, el algoritmo de Euclides es útil para obtener el mcd de dos números. ¿Cómo proceder en caso de requerir el máximo común divisor de más de dos enteros? El mcd de n enteros $a_1, a_2, \ldots, a_n$ se define como $mcd(a_1, mcd(a_2, \ldots, mcd(a_{n-1}, a_n)))$.

Ejemplo 82. ¿Cuál es el máximo común divisor de tres números enteros dados a, b, c?

> **Solución.** Primero habrá que obtener el mcd de a y b, para luego obtener el máximo común divisor de él y de c.

```
1   int A_Euclides(int num1, int num2)
2   {
3       int r;
4       r = num1;
5       do
6       {
7           num1 = num2;
8           num2 = r;
9           r = num1 % num2;
10      } while (r != 0)
11
12      return num2;
13  }
```

```
14
15  int solucion()
16  {
17      int a, b, c, mcd;
18      cin>>a>>b>>c;
19
20      mcd = A_Euclides(a, b);
21      mcd = A_Euclides(mcd, c);
22
23      return mcd;
24  }
```

Ejemplo 83. Considera N listones de medidas $n_1, n_2, \ldots, n_N$, todos números naturales. Si queremos dividir todos los listones en segmentos de la misma longitud (entera), de tal forma que se requiera la mínima cantidad de cortes, ¿cuánto deberán medir dichos pedazos?

> **Solución.** Para minimizar la cantidad de cortes hay que minimizar la cantidad de segmentos de listones, lo que implica que hay que maximizar el tamaño de los listones, es decir, hay que encontrar el máximo común divisor de todos los n_i, $i = 1, 2 \ldots, N$.

```
1   int solucion()
2   {
3       int N, n, i, medida;
4       vector<int> listones;
5       cin>>N;
6       for (i = 0; i < N; i = i + 1)
7       {
8           cin>>n;
9           listones.push_back(n);
10      }
11
12      medida = A_Euclides(listones[0], listones[1]);
13      for (i = 2; i < N; i = i + 1)
14          medida = A_Euclides(medida, listones[i]);
15
16      return medida;
17  }
```

Ejemplo 84. Considera un terreno de medidas $m \times n$, en donde m y n son enteros positivos dados. El mencionado terreno pronto será reforestado. Para esto, se debe seguir un protocolo, que consiste en dividirlo en parcelas cuadradas y, en cada una, plantar exactamente k árboles.

Como hay que prevenir que los árboles plantados excedan la capacidad de brindar nutrientes del suelo, y también hay que ahorrar dinero, se busca comprar la mínima cantidad de árboles posible, de tal forma que se respete el protocolo.

Encuentra cuáles deben ser las dimensiones de las parcelas y el número de árboles que serán necesarios para todo el terreno.

> **Solución.** Para minimizar la cantidad de árboles, hay que minimizar las parcelas en las que se partiocionará el terreno. Como éstas deben ser cuadradas, habrá que encontrar un número t tal que sea divisor de m y de n, y que, además, sea el máximo entero que cumpla con esa característica.
>
> Así, tendremos $(\frac{m}{t})(\frac{n}{t})$ parcelas y requeriremos $(\frac{m}{t})(\frac{n}{t})k$ árboles.

```
1  void solucion()
2  {
3      int m, n, k, t, arboles;
4      cin>>m>>n>>k;
5
6      t = A_Euclides(m, n);
7      arboles = (m/t) * (n/t) * k;
8
9      cout<<"Medidas: "<<t<<"; árboles: "<<arboles<<"\n";
10 }
```

Ejemplo 85. Dada una fracción $\frac{a}{b}$, determina si ésta es irredubile.

En caso de que sea reducible, devuelve el número de veces que la fracción puede ser reducida dividiendo el numerador y el denominador entre su máximo común divisor. Por ejemplo, dada la fracción $\frac{10}{12}$, encontrarás que $mcd(10, 12) = 2$, por lo que ésta es reducible a $\frac{5}{6}$, que ya no puede ser simplificada, por tanto únicamente pudimos reducir la fracción dada una vez.

Solución. Una fracción únicamente es reducible cuando su numerador y su denominador tienen factores en común distintos de 1. De esta forma, ambos se podrán dividir para obtener una equivalencia de la fracción dada con enteros más pequeños. Así, sabremos que si $mcd(a, b) = 1$, entonces no habrá divisores que compartan a y b y la fracción no podrá reducirse.

Por otro lado, hay que preguntarnos cuántas veces podremos reducir nuestra fracción en caso de que el máximo común divisor de a y b sea distinto de 1. Si $d = mcd(a, b)$, existen enteros a_1 y b_1 para los que $a = d \cdot a_1$ y $b = d \cdot b_1$. De aquí, deducimos que a_1 y b_1 son primos relativos, pues si no lo fueran a y b tendrían un mcd mayor a d. Así, los enteros $a_1 = a/d$ y $b_1 = b/d$ son primos relativos, con lo que concluimos que:

$$mcd\left(\frac{a}{d}, \frac{b}{d}\right) = 1$$

Lo que nos dice que, dada una fracción reducible $\frac{a}{b}$, únicamente es posible dividir su numerador y su denominador una vez entre d, luego, la fracción resultante será irreducible.

```
1  void solucion()
2  {
3      int a, b, d;
4      char c;
5      cin>>a>>c>>b;
6
7      d = A_Euclides(a, b);
8      if (d != 1)
9          cout<<1<<"\n";
10     else
11         cout<<"Irreducible\n";
12 }
```

2.5.2. Algoritmo de Euclides Extendido

El Algoritmo Extendido de Euclides, además de encontrar el máximo común divisor de dos enteros, permite expresarlo como una combinación lineal de ellos, como lo enuncia el siguiente lema.

Lema 1. [Lema de Bezout]. Sean a y b dos números enteros. El máximo común divisor de a y b se puede escribir como la menor combinación lineal positiva $ax + by$, donde $x, y \in \mathbb{Z}$.

Demostración. Sea d el máximo común divisor de a y b, y sea $m = ax + by$ la menor combinación lineal positiva de a y b. Como $d \mid a$ y $d \mid b$, sabemos que $d \mid m$, por lo que $d \leq |m|$.

Demostremos ahora que $m \mid a$ y $m \mid b$. Procederemos por contradicción y supondremos que $m \nmid a$ para empezar, por lo que la división $a \div m$ no puede tener un residuo 0. Por el algoritmo de la división, sabemos que a también se puede expresar como $a = qm + r$, en donde $0 < r < m$. Despejando,

$$\begin{aligned} r &= a - q \cdot m \\ &= a - q(ax + by) \\ &= a - qax - qby \\ &= (1 - qx)a + (-qy)b \end{aligned}$$

Lo que prueba que r es también una combinación lineal de a y b; pero r es una combinación lineal positiva y m es la menor de ellas, por lo que $m \leq r$ lo que nos lleva a una contradicción que implica que nuestra suposición inicial ($m \nmid a$) no era cierta y que a sí es divisible por m. Así, $m \mid a$. Análogamente se demuestra que $m \mid b$.

Ya que d divide tanto a a como a b, dividirá a cualquier combinción lineal de ellos, en particular, a m, lo que implica que d debe ser menor o igual que m, y dado que d es el máximo divisor común y no puede haber ninguno menor que él, sólo puede pasar que $d = m$.

□

El teorema anterior garantiza la existencia de los enteros x y y; ahora la pregunta es ¿cómo obtenemos los valores de x y y? En realidad, será muy sencillo tomando en cuenta que ya conoces el algoritmo de Euclides.

En el teorema 14 vimos que, aplicando repetidamente el algoritmo de la división, en cada iteración obtendremos un $r_i = r_{i+1}q_{i+1} + r_{i+2}$, $0 \leq i \leq n-1$, por lo que en la penúltma iteración tendremos $r_{n-2} = r_{n-1}q_{n-1} + r_n$. Despejando,

$$r_n = r_{n-2} - r_{n-1}q_{n-1} \tag{2.1}$$

Luego, podemos escribir a r_{n-1} en función de r_{n-2} y r_{n-3}, pues de la iteración antepenúltima, lo podemos despejar como $r_{n-1} = r_{n-3} - r_{n-2}q_{n-2}$. Sustituyendo r_{n-1} en 2.1:

$$\begin{aligned} r_n &= r_{n-2} - r_{n-1}q_{n-1} \\ &= r_{n-2} - (r_{n-3} - r_{n-2}q_{n-2})q_{n-1} \end{aligned}$$

Siguiendo bajo la misma lógica:

$$\begin{aligned} r_n &= r_{n-2} - r_{n-1}q_{n-1} \\ &= r_{n-2} - (r_{n-3} - r_{n-2}q_{n-2})q_{n-1} \\ &= [r_{n-4} - r_{n-3}q_{n-3}] - [(r_{n-3} - (r_{n-4} - r_{n-3}q_{n-3})q_{n-2})q_{n-1}] \\ &= \cdots \end{aligned}$$

El proceso se repetirá hasta que todos los residuos hayan sido escritos en función de r_0 y de r_1 (es decir, de a y b), en donde habremos obtenido los valores de x y y.

Encontremos, por ejemplo, el máximo común divisor de 102 y 38 como una combinación lineal de los mismos.

Primero, aplicamos el algoritmo de Euclides antes visto.

a	b	$r = a \,\%\, b$
102	38	26
38	26	12
26	12	**2**
12	2	0

Ahora iremos a la inversa:

$$\begin{aligned} 2 &= 26 - 2(12) \\ &= 26 - 2[38 - 1(26)] = 3(26) - 2(38) \\ &= 3[102 - 2(38)] - 2(38) = 3(102) - 8(38) \end{aligned}$$

Por lo tanto, $x = 3$ y $y = -8$.

Ejemplo 86. Obtén los valores de x y y para los que el máximo común divisor de dos enteros dados a y b es $ax + by$.

Solución. Aplicando el algoritmo extendido de Euclides:

```
int x, y, mcd;

void Euclides_Extendido(int a, int b)
{
    int x1, y1;

    if (b == 0)
    {
        x = 1;
        y = 0;
        mcd = a;
        return;
    }

    Euclides_Extendido(b, a % b);

    //Sustitución de los r's.
    x1 = y;
    y1 = x - (a / b) * y;
    x = x1;
    y = y1;
}

void solucion()
{
    int a, b;
    cin>>a>>b;
    Euclides_Extendido(a, b)
    cout<<x<<" "<<y<<"\n";
}
```

Ejemplo 87. Imagina un estacionamiento de una sola fila en el que los cajones están numerados de $-\infty$ a $+\infty$. Para estacionarte, comienzas a la altura del cajón 0 y puedes avanzar y/o retroceder exactamente a o b cajones tantas veces como requieras. ¿Puedes garantizar que podrás estacionarte en todos los cajones pares? (Problema modificado de "Hurgando en el CEDETEC", tomado de [3]).

Solución. Sin importar cuántas veces decidas ir hacia adelante o hacia atrás, la igualdad $ak + bm = 2$, en donde k son los "pasos" de tamaño a y m los de tamaño b que das, representará el escenario que posibilita que todo espacio par sea alcanzable. Hay que señalar que, si k es positivo, implica un avance, mientras que si es negativo, implica un retroceso en la recta numérica. Lo mismo ocurre con m.

Ya vimos que el máximo común divisor de dos enteros se puede expresar como una combinación lineal en función de dichos enteros, y justo es lo que aprovecharemos aquí; entonces, requerimos que el mcd de a y b sea par, pero eso no es todo, requerimos que el mcd sea 2, pues de esa forma garantizaremos que es posible pasar por todos los cajones múltiplos de 2 (y no sólo de 4, 6, 8,...).

Así, lo que tendremos que hacer será ejecutar el algoritmo de Euclides sobre los números a y b, y comprobar que su máximo común divisor sea 2, pues será el único caso en que estaremos seguros de que cada cajón par es alcanzable.

```
1  bool solucion()
2  {
3      int a, b, mcd;
4      cin>>a>>b;
5
6      mcd = A_Euclides(a, b);
7      if (mcd == 2)
8          return true;
9      else
10         return false;
11 }
```

Ejemplo 88. Siguiendo sobre el problema anterior, ¿qué requerimos para garantizar que absolutamente todos los espacios para estacionarse sean alcanzables?

> **Solución.** Aplicando el razonamiento del problema anterior, necesitamos que el mcd de a y b sea 1, es decir, que a y b sean primos relativos.

```
bool solucion()
{
    int a, b, mcd;
    cin>>a>>b;

    mcd = A_Euclides(a, b);
    if (mcd == 1)
        return true;
    else
        return false;
}
```

Cuando dos números son primos relativos, el algoritmo extendido de Euclides tiene una utilidad especial. Veamos por qué.

Antes que nada, recordemos que el inverso multiplicativo de un número a es algún número a' tal que $a \cdot a'$ es igual al neutro multiplicativo, esto es 1. En aritmética modular, en particular, tanto a como a' deben ser enteros y su producto nuevamente debe ser 1, sólo que este 1 será sobre la clase de equivalencia módulo n. Un ejemplo de inversos multiplicativos son 2 y 6, módulo 11, pues su producto será $2(6) \equiv 12 \equiv 1 \ (mod\ 11)$.

Pensemos entonces en dos enteros a y b que son primos relativos, lo que implica que $mcd(a, b) = 1 = ax + by$. Despejando, $ax = 1 - by$, de lo que se sigue que $ax \equiv 1 \ (mod\ b)$. Esto nos dice que x es el inverso multiplicativo de a, módulo b. En este punto llegamos a algo interesante, y es que *un número únicamente puede tener un inverso multiplicativo, módulo b, si es primo relativo de a.*

Nota que, cuando estudiamos las maneras de aplicar aritmética modular con divisiones involucradas, en realidad lo que hicimos fue hallar el inverso modular multiplicativo del denominador, pues nuestros resultados se basan en el cumplimiento

del Pequeño Teorema de Fermat. Así pues, el algoritmo extendido de Euclides, nos brinda una forma más para aplicar aritmética modular cuando hay divisiones involucradas.

Ejemplo 89. Para entenderlo mejor, apliquemos este procedimiento al problema 70, en el que hay que determinar el producto de un subarreglo $[m, n]$ de un arreglo de enteros de tamaño 10^5, y regresar su equivalente módulo 71.

> **Solución.** Cuando resolvimos por primera vez este problema, vimos que el producto del intervalo $[m, n]$, módulo 71, se obtendrá a través de la expresión:
>
> $$P_{m,n} \,\%\, 71 = \frac{P_{0,n}}{P_{0,m-1}} \,\%\, 71$$
>
> Entonces, buscamos el inverso modular multiplicativo de $P_{0,m-1}$, módulo 71. Para esto, aplicaremos el algoritmo extendido de Euclides tomando como a a $P_{0,m-1}$ y como b a 71.

```
int solucion()
{
    int m, n, mod = 71;
    cin>>m>>n;

    Euclides_Extendido(P[m - 1], mod);
    if (mcd != 1 )
    {
        cout<<"No se puede.\n";
        return -1;
    }
    return P[n] * x % mod;
}
```

El problema con este método es que hay que asegurarnos de que a sea primo relativo de b para cualquier entrada dada, lo que podría ser realmente complicado.

2.6. Mínimo Común Múltiplo

El mímimo común múltiplo de dos enteros a y b es el menor de los múltiplos positivos que comparten dichos números. Éste se denota como $[a, b]$, o bien, como $mcm[a, b]$ y $lcm[a, b]$ por sus siglas en inglés (lowest common multiple). Esta obra en particular hará uso de la notación $mcm[a, b]$.

La obtención del mínimo común múltiplo de dos números es fácil de obtener "a mano", como lo recuerda el siguiente ejemplo, en el que buscamos el mcm de 12 y 50, haciendo la descomposición por primos simultánea de ambos números.

$$\begin{array}{rr|l} 12 & 50 & 2 \\ 6 & 25 & 2 \\ 3 & 25 & 3 \\ 1 & 25 & 5 \\ 1 & 5 & 5 \\ 1 & 1 & \end{array}$$

Así, tomando todos los factores resultantes, sabemos que el $mcm[12, 50] = 2^2 \cdot 3 \cdot 5^2 = 300$.

Formalizando el conocimiento que seguramente obtuviste en educación básica, el mínimo común múltiplo de dos enteros a y b, mismos que se pueden expresar como $p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdots p_k^{\alpha_k}$ y $p_1^{\beta_1} \cdot p_2^{\beta_2} \cdots p_k^{\beta_k}$, respectivamente, es:

$$mcm[a, b] = p_1^{M_1} \cdot p_2^{M_2} \cdot \cdots \cdot p_k^{M_k}$$

En donde cada M_i es el máximo de α_i y β_i.

Nota que puede suceder que a y b no compartan algún factor primo, pero esa situación se solucionará al darse cuenta de que el mencionado primo se puede agregar sin modificar una descomposición canónica siempre que tenga exponente 0.

La pregunta ahora es ¿cómo obtener el mínimo común múltiplo de dos números a través de un algoritmo computacionalmente eficiente? Los siguientes resultados mostrarán que, una vez obtenido el máximo común divisor de a y b, la obtención del

mínimo común múltiplo es sumamente sencilla. Ya que el mcm de a y b es el mismo que el de a y $-b$, todos los resultados se trabajarán sobre enteros no negativos.

Proposición 23. Sean a y b dos enteros y t un número para el que $a \cdot t$ y $b \cdot t$ son enteros también. Entonces:

$$t \cdot mcm[a, b] = mcm[t \cdot a, t \cdot b]$$

Demostración. Si $a = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdots p_k^{\alpha_k}$ y $b = p_1^{\beta_1} \cdot p_2^{\beta_2} \cdots p_k^{\beta_k}$,

$$mcm[a, b] = p_1^{M_1} \cdot p_2^{M_2} \cdots p_k^{M_k}$$

En donde cada M_i es el máximo de entre α_i y β_i. Entonces,

$$t \cdot mcm[a, b] = t \cdot \left(p_1^{M_1} \cdot p_2^{M_2} \cdots p_k^{M_k}\right)$$

Por otro lado, $t \cdot a = t \cdot p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdots p_k^{\alpha_k}$ y $t \cdot b = t \cdot p_1^{\beta_1} \cdot p_2^{\beta_2} \cdots p_k^{\beta_k}$, de lo que se sigue:

$$mcm[t \cdot a, t \cdot b] = Max\{t, t\} \cdot p_1^{M_1} \cdot p_2^{M_2} \cdots p_k^{M_k}$$

$$= t \cdot p_1^{M_1} \cdot p_2^{M_2} \cdots p_k^{M_k}$$

Por lo tanto,

$$t \cdot mcm[a, b] = mcm[t \cdot a, t \cdot b] = t \cdot p_1^{M_1} \cdot p_2^{M_2} \cdot \cdots \cdot p_k^{M_k}$$

□

Proposición 24. Sean a y b dos números enteros distintos de 0. Su mínimo común múltiplo es:

$$mcm[a, b] = \frac{|a \cdot b|}{mcd(a, b)}$$

Demostración. Pensemos primero en el caso en el que $mcd(a,b) = 1$: Sea m el mínimo común múltiplo de a y b; entonces existe algún entero k para el que $m = a \cdot k$, y ya que m es divisible por b, $b \Big| a \cdot k$. Del hecho de que a y b son primos relativos, sabemos que b no puede dividir a a y por tanto $b \Big| k$. Luego, $b \leq k$ y $a \cdot b \leq a \cdot k = m$, pero ya que m es el mínimo común múltiplo de a y b, no puede pasar que $a \cdot b < m$, con lo que el único hecho posible será que $m = a \cdot b$. De esta forma,

$$mcm[a,b] = a \cdot b = 1 \cdot (ab) = mcd(a,b) \cdot |a \cdot b|$$

Es decir, dados dos enteros a y b que son primos relativos, se cumplirá que su mínimo común múltiplo es $|a \cdot b|$.

Veamos ahora el caso en el que a y b no son primos relativos. Llamemos d al máximo común divisor de a y b. Podemos expresar a a y b como $a = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdots p_k^{\alpha_k}$ y $b = p_1^{\beta_1} \cdot p_2^{\beta_2} \cdots p_k^{\beta_k}$; como $\frac{a}{d}$ y $\frac{b}{d}$ son enteros, por la proposición 23 podemos asegurar:

$$mcd\Big(\frac{a}{d}, \frac{b}{d}\Big) = \frac{1}{d} \cdot mcd(a,b) = \frac{1}{d} \cdot d = 1$$

Tenemos entonces que $\frac{a}{d}$ y $\frac{b}{d}$ son primos relativos, por lo que:

$$mcd\Big(\frac{a}{d}, \frac{b}{d}\Big) = 1 \quad \text{y} \quad mcm\Big[\frac{a}{d}, \frac{b}{d}\Big] = \Big|\frac{a}{d} \cdot \frac{b}{d}\Big|$$

Multiplicando ambos lados de la segunda expresión por d^2:

$$d^2 \cdot mcm\Big[\frac{a}{d}, \frac{b}{d}\Big] = d^2 \cdot \frac{|a \cdot b|}{d^2}$$

Por la proposición 23, $d^2 \cdot mcm\Big[\frac{a}{d}, \frac{b}{d}\Big] = d \cdot mcm[a,b]$, de lo que se sigue:

$$d \cdot mcm[a,b] = mcd(a,b) \cdot mcm[a,b] = 1 \cdot |a \cdot b|$$

$$\Longrightarrow mcm[a,b] = \frac{|a \cdot b|}{mcd(a,b)}$$

□

Nuevamente, este algoritmo proporciona el mínimo común múltiplo de únicamente dos enteros. Para obtener el mcm de tres números o más, sólo hay que considerar que $mcm[a_1, a_2, \ldots, a_n] = mcm[a_1, mcm[a_2, \ldots, mcm[a_{n-1}, a_n]]]$.

Ejemplo 90. Obtén el mínimo común múltiplo de dos enteros dados a y b.

Solución. Aplicando la fórmula obtenida arriba:

```
int obtener_mcm()
{
    int a, b, mcd, mcm;
    cin>>a>>b;

    mcd = A_Euclides(a,b);
    mcm = abs(a * b) / mcd;

    return mcm;
}
```

Ejemplo 91. Imagina un engranaje de tres engranes que forman parte de cierto mecanismo. Los engranes tardan t_1, t_2 y t_3 segundos en dar una vuelta completa cada uno (t_1, t_2 y t_3 son enteros positivos). ¿En cuántos segundos las tres ruedas se encontrarán por octava vez, todas, en su posición de inicio?

Solución. Nombremos A, B y C a los engranes que tardan t_1, t_2 y t_3 segundos, respectivamente. La primera vez en que el engranaje completo regrese a su posición original, el engrane A habrá dado v_1 vueltas completas, el engrane B v_2 y el engrane C v_3, para $v_1, v_2, v_3 \in \mathbb{N}$. Estamos buscando entonces los valores para los que se cumplirá que $v_1t_1 = v_2t_2 = v_3t_3$ por octava vez, esto es, el mínimo común múltiplo de t_1, t_2, t_3 multiplicado por 8.

Invocaremos la función del programa pasado para obtener el mínimo común múltiplo de dos enteros.

```
int solucion()
{
```

```
3       int t1, t2, t3, mcd, mcm;
4       cin>>t1>>t2>>t3;
5
6       mcm = obtener_mcm(t1, t2);
7       mcm = obtener_mcm(mcm, t3);
8
9       return 8 * mcm;
10  }
```

Ejemplo 92. Una pista de atletismo profesional tiene una longitud de 400 metros; dos personas, A y B han estado entrenando juntas. La persona A corre m_A metros por minuto, y la persona B avanza m_B, también por minuto, donde m_A y m_B son dos enteros positivos distintos. ¿Cuántos minutos habrán pasado antes de que ambos individuos se encuentren simultáneamente en el inicio de la pista por primera vez, si requerimos que los dos hayan corrido minutos enteros? ¿Cuántas vueltas habrá dado cada corredor?

Solución. Para que A corra minutos enteros, es necesario que la cantidad de metros que recorra sea múltiplo de m_A, y ya que estamos buscando el momento en que llega al punto de inicio, también debe ser múltiplo de 400; así pues, requerimos el primer momento en que ambos múltiplos coincidan, es decir, el mínimo común múltiplo de m_A y 400, nombrémoslo MCM_A. Bajo la misma lógica, también requeriremos buscar el mcm de m_B y 400: MCM_B.

Lo que hemos hecho hasta ahora nos dice los metros que debe recorrer cada competidor para llegar al punto de inicio por su cuenta, en minutos enteros, es decir, estos hechos no necesariamente ocurren simultáneamente. Para ello, nota que debemos encontrar el primer momento en el que coincidan, esto es, el mcm de MCM_A/m_A y MCM_B/m_B.

Finalmente, habrá que multiplicar nuestra respuesta por m_A para obtener los metros que A recorrió, dato que, a su vez, dividiremos entre 400, para obtener el número de vueltas enteras al que ese número equivale; el mismo proceso aplicaremos para B.

```
1  void solucion()
2  {
3      int mA, mB, mcmA, mcmB, mcm, vueltas_A, vueltas_B;
4      cin>>mA>>mB;
5
6      mcmA = A_Euclides(mA, 400);
7      mcmB = A_Euclides(mB, 400);
8      mcm = A_Euclides(mcmA/mA, mcmB/mB);
9
10     vueltas_A = mcm * mA / 400;
11     vueltas_B = mcm * mB / 400;
12     cout<<mcm<<", "<<vueltas_A<<", "<<vueltas_B<<"\n";
13 }
```

Ejemplo 93. Una máquina barajadora utiliza, en todas sus iteraciones, e l mismo algoritmo para revolver las cartas. Para nuestros fines, en este problema imaginaremos un conjunto de barajas numeradas. Suponiendo que la configuración inicial siempre es $1, 2, 3, \ldots, n$, y dado el segundo acomodo que adopta el mazo, ¿cuántas rondas es necesario que haga la máquina para devolver a las cartas a su posición inicial?

Por ejemplo, si tenemos un mazo de 5 cartas cuya segunda configuración corresponde a 4 3 2 5 1, sabemos que la primera carta pasó al último lugar, la segunda pasó al tercer lugar, la tercera al segundo, la cuarta al primero y la quinta al cuarto; para la siguiente iteración, nuevamente la primera carta (en este punto, la primera carta es 4) pasará al último lugar, la segunda al tercero, la tercera al segundo, la cuarta al primero y la quinta al cuarto.

> **Solución.** Pensando en almacenar las n cartas en un vector, por comodidad, hagámoslo en uno de tamaño $n + 1$, e ignoremos la posición 0.
>
> Comencemos analizando el ejemplo que nos han proporcionado:

Carta	Posición
1	5
2	3
3	2
4	1
5	4

En particular, la carta con el número 1 pasó a la posición 5, para luego ocupar el lugar 4 y volver al 1; por su parte, la carta con el número 2 pasa a la tercera posición y regresa a la segunda. De esta forma, tenemos dos ciclos de permutaciones: 1-5-4-1 y 2-3-2. En el momento en que vuelve a aparecer el número con el que un ciclo inició, sabemos que dicho ciclo comienza de nuevo. Así, lo que nos interesa es conocer el momento en que todos los ciclos de permutaciones presentes en el algoritmo de la máquina barajadora vuelven al inicio, es decir, a su posición original, esto es, el mínimo común múltiplo de todos los tamaños de los ciclos de permutaciones encontrados.

```
1  int solucion()
2  {
3      int n, a, i, inicio, tam, mcm;
4      vector<int> posicion, marcado, ciclos;
5      cin>>n;
6      posicion.resize(n + 1); marcado.resize(n + 1);
7
8      for (i = 1; i <= n; i = i + 1)
9      {
10         cin>>a;
11         posicion[a] = i;
12     }
13
14     //Encontrar ciclos de permutaciones.
15     for (i = 1; i <= n; i = i + 1)
16     {
17         if (marcado[i])
18             continue;
19
20         inicio = i;
21         a = i;
22         tam = 1;
23         while (posicion[a] != inicio)
```

```
24          {
25              a = posicion[a];
26              tam = tam + 1;
27          }
28          ciclos.push_back(tam);
29      }
30
31      //Para MCM.
32      mcm = ciclos[0];
33      for (int e:ciclos)
34          mcm = obtener_mcm(mcm, e);
35      return mcm;
36  }
```

2.7. Exponenciación Rápida

Existen diversos algoritmos que aprovechan la propiedad binaria de las computadoras. En particular, cuando tratamos el tema de las divisiones dentro de la aritmética modular (y en otras varias ocasiones), mencionamos algunos métodos que requieren la obtención de una potencia muy grande; abordamos dichos ejemplos bajo la premisa de que aprenderíamos una manera de realizar potenciaciones de forma logarítmica en lugar de realizar una multiplicación de cierto número por sí mismo k veces.

Hay que mencionar que una de las ventajas más notables de la *exponenciación rápida* es que no es necesario realizar el cambio de base, pues la computadora lo hará automáticamente, como ya lo mencionamos.

Comencemos entonces con el método: Sea n cualquier número y k un entero positivo. Si pensamos en la expresión n^k, podemos expresarla como $n^k = n^{x_0} \cdot n^{x_1} \cdot n^{x_2} \cdots n^{x_r}$, en donde $\sum_{i=0}^{r} x_i = k$.

Por otro lado, como ya dijimos, la computadora entiende al número k como un número en base 2, lo que implica que k es lo mismo que $x_r(2^r) + x_{r-1}(2^{r-1}) + \ldots +$

$x_1(2)+x_0(2^0)$, en donde cada x_i es 0 o 1, y $r+1$ es el número de dígitos de k en base 2.

Así, podemos decir que:

$$\begin{aligned} n^k &= n^{[x_r(2^r)+x_{r-1}(2^{r-1})+\cdots+x_1(2^1)+x_0(2^0)]} \\ &= n^{x_r(2^r)} \cdot n^{x_{r-1}(2^{r-1})} \cdot \ldots \cdot n^{x_1(2^1)} \cdot n^{x_0(2^0)} \end{aligned}$$

Y dado que x_i únicamente puede tomar valores de 0 o 1, solamente será necesario obtener n^{2^i}, y decidir si se tomará o no (en lo sucesivo lo comprenderás mejor con un ejemplo).

Ya comprendimos qué es lo que se busca hacer con la exponenciación rápida, veamos ahora cómo llegar a ello. Si la potencia que nosotros tenemos (en base decimal) es impar, entonces, el último dígito del mismo número en base 2 es 1, lo que implica que esa potencia sí se utilizará. En otras palabras, si $k \,\%\, 2 = 1$, entonces se usa n^1, de lo contrario no se considera; independientemente de que la potencia n^1 se haya tomado o no, se guardará, pues la necesitaremos para calcular n^2 al elevarla al cuadrado. Nota que el dígito que nos interesa en este punto ya no es el último de la representación binaria de k, sino el penúltimo. Para obtenerlo, bastará con dividir a k entre 2, acción que ocasionará la eliminación del que antes era el último dígito (en binario), y repetiremos el proceso: Si $k = k/2$ es impar, el último dígito es 1 y por tanto se debe usar esa potencia, de lo contrario, únicamente pasaremos al análisis del siguiente dígito. El procedimiento se repetirá hasta que k sea 0.

Para comprender mejor el algoritmo, planteémoslo sobre un ejemplo. Digamos que deseamos obtener 7^9. Para comenzar, veamos cuál es la representación binaria de 9:

1	0	0	1

Lo anterior simplemente nos servirá como un elemento ilustrativo para comprender la forma en que la computadora realiza este procedimiento.

Lo primero será preguntar si el número 9 es par; ya que no lo es, entonces sí se utilizará la potencia 7^1. Luego, dividimos $9 \div 2$, y ya que la computadora se quedará con la parte entera, ahora analizamos el número 4 (100 en binario) y preguntamos si es par. Como lo es, el último dígito de su representación es 0 y no se usará, pero

sí será necesario guardar 7^2. Nuevamente, dividimos $4 \div 2 = 2$ (10 en binario), y ya que es par, no se usa la potencia 7^4. Finalmente, $2 \div 2 = 1$, y como 1 es impar, sí requeriremos el número 7^8. Dado que $1 \div 2 = 0$, con esto terminamos el ejemplo, y concluimos que $7^9 = 7^1 \cdot 7^8$. De esta forma, sólo nos ocupamos en calcular 7^2, 7^4 y 7^8 multiplicando el anterior por sí mismo, en lugar de realizar 9 multiplicaciones.

Ejemplo 94. Dados un número real n y un entero positivo k, determina n^k.

Solución. Aplicando directamente el algoritmo de exponenciación rápida:

```
long long exp_rapida(long long n, long long k)
{
    long long respuesta = 1;

    while (k! = 0)
    {
        if (n % 2 == 1)
            respuesta = respuesta * n;

        n = n * n;
        k = k / 2;
    }

    return respuesta;
}
```

En varios ejercicios de la subsección *Divisiones en Aritmética Modular* necesitamos la resolución de una potencia muy grande, a la que, además, había que aplicarle algún módulo. El siguiente ejemplo, entonces, será justo la aplicación del algoritmo de exponenciación rápida para la realización de potencias con la inclusión de módulos.

Ejemplo 95. Dados dos enteros positivos n y k, determina el menor valor no negativo de $n^k (mod\ 787)$.

Solución. Recordemos que, para que la aplicación de módulo sea efectiva, es necesario que, posterior a cada operación realizada, se obtenga la equivalencia modular; recordemos también que siempre que empleamos la operación %, se obtiene el menor de los enteros en la clase modular con el

mismo signo que el número modulado, y ya que únicamente trabajamos con números naturales, no tenemos que preocuparnos por contemplar los casos en los que el resultado es negativo.

Así pues, únicamente será necesaria la aplicación del algoritmo de exponenciación rápida intercalando la operación %.

```
long long ExpRapida_modulo(long long n, long long k, int mod)
{
    long long respuesta = 1;

    while (k!= 0)
    {
        if (n % 2 == 1)
            respuesta = respuesta * n % mod;

        n = n * n % mod;
        k = k / 2;
    }
    return respuesta;
}

void solucion()
{
    long long n, k;
    cin>>n>>k;
    cout<<ExpRapida_modulo(n, k, 787);
}
```

Ejemplo 96. Dado un entero n y un número natural k, determina el mínimo valor no negativo de:

$$412 \cdot k^k - 112 \cdot n^k \ (mod\ 787)$$

Solución. En esta ocasión, tenemos la presencia de una resta, lo que implica un riesgo de que el resultado de aplicar la operación % sea negativo. Dado que queremos que nuestro resultado sea no negativo, deberemos sumar 787 en cuanto se aplica la substracción.

```
int solucion()
{
```

```
3       long long n, k;
4       int mod = 787, respuesta;
5       cin>>n>>k;
6
7       //Resolviendo la expresión por partes.
8       respuesta = 412 * ExpRapida_modulo(k, k, mod);
9       respuesta = ((respuesta - 112 * ExpRapida_modulo(n, k, mod))
10                  % mod + mod) % mod;
11
12      return respuesta;
13  }
```

Vista esta técnica, únicamente será necesario reemplazar la función *potencia_modulo* por *ExpRapida_modulo*.

Para finalizar el tema, nota que es claro que la complejidad de obtener n^k con el método de exponenciación rápida es $O(log_2 k)$, que obviamente es menor que $O(k)$, lo que resultaría de obtener la k-ésima potencia de n de la manera tradicional.

A lo largo de este capítulo abordamos una gran variedad de temas, tratando siempre de incluir ejemplos que muestren la utilidad de la teoría mostrada, sin embargo, es probable que requieras o prefieras practicar con más de ellos. En [17] tendrás acceso a este tipo de temas, que, si bien se dan desde un punto de vista puramente matemático, te ayudará a ampliar tu conocimiento y a retar tu conocimiento, tus habilidades y tu creatividad.

3

Teoría de Grafos

Antes de comenzar con este capítulo, es importante mencionar que éste tomará un camino más bien teórico, tratando de explicar las bases de los algoritmos comúnmente usados en programación competitiva y la razón de su funcionamiento; aunque el enfoque que aquí se abordará no está peleado con la implementación de los algoritmos, no abundaremos en ejemplos como se ha hecho en los capítulos anteriores, no obstante, si acudes a [3], a [18], y a [19], encontrarás numerosos ejemplos que ilustrarán desde un punto de vista computacional práctico lo que aquí trataremos.

3.1. Conceptos Básicos

Comencemos viendo algunas definiciones básicas.

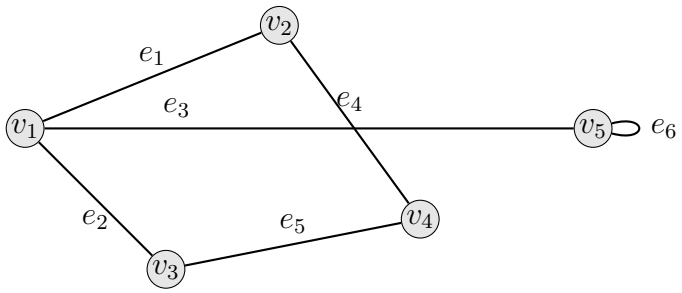
Definición 1. Un grafo G es una tripleta ordenada (V, E, ψ), en donde V es un conjunto de vértices, E es un conjunto de aristas y ψ es una función tal que $\psi : E \to \binom{[V]}{1} \cup \binom{[V]}{2}$, lo que, en términos más simples, implica que los vértices podrán ser relacionados a través de aristas con otros vértices y consigo mismos.

Definición 2. Se dice que una arista e es *incidente* a un vértice v si al menos un extremo de e se encuentra en v.

Si ambos extremos de e pasan por v, e se conoce como un *bucle*.

Definición 3. Se dice que un vértice u es *adyacente* a un vértice v si existe una arista e cuyos extremos se encuentran en u y en v. La arista e puede ser denotada también como uv y como (u, v).

Ejemplo 97. Sea $G = (V, E, \psi)$ el grafo mostrada a continuación. ¿Cómo se define cada elemento de dicha tripleta?



Solución. V, E y ψ quedarían definidos de la siguiente manera:

$$
\begin{aligned}
V &= \{v_1, v_2, v_3, v_4, v_5\} \\
E &= \{e_1, e_2, e_3, e_4, e_5, e_6\} \\
\psi(e_1) &= \{v_1, v_2\}, \\
\psi(e_2) &= \{v_1, v_3\}, \\
\psi(e_3) &= \{v_1, v_5\}, \\
\psi(e_4) &= \{v_2, v_4\}, \\
\psi(e_5) &= \{v_3, v_4\}, \\
\psi(e_6) &= \{v_5\}
\end{aligned}
$$

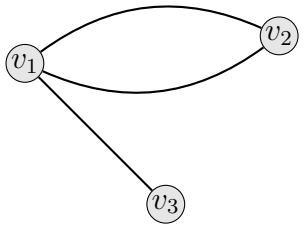
Definición 4. Un grafo G es *finito* si los conjuntos V y E de la tripleta (V, E, ψ) son finitos.

Definición 5. Un grafo que cumple que $E = \emptyset$ se llama *vacío*.
Un grafo que cumple que $V = \emptyset$ se llama *nulo*.

Un grafo que cumple que $|V|$ [1] $= 1$ se llama *trivial*.

Definición 6. Se dice que un grafo G es un grafo *simple* si ninguna de sus aristas es un bucle y si no existe más de una arista que incida en el mismo par de vértices.

Ejemplo 98. La gráfica mostrada a continuación no es una gráfica simple, pues existen dos aristas incidentes a v_1 y v_2.



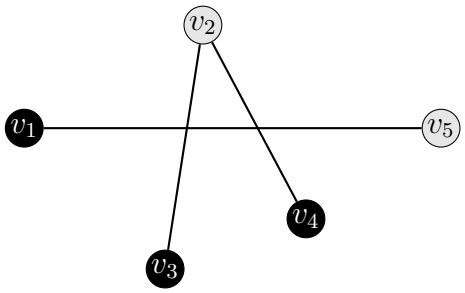
Definición 7. Llamamos *grafo completo* a un grafo simple en el que, para todo $u, v \in V$, $u \neq v$, existe exactamente una arista e incidente en u y en v, en otras palabras, en un grafo completo existe exactamente una arista que une a cada par de vértices.
Claramente, existe únicamente un grafo completo para cada valor distinto de $|V|$, salvo los isomorfismos[2]. Si $|V| = n$, entonces podemos utilizar la notación K_n.

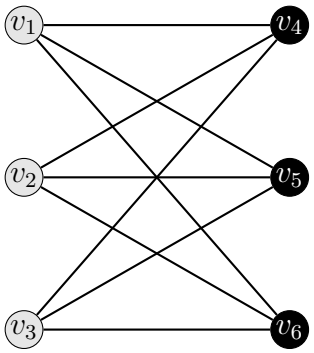
Definición 8. Un grafo $k-$*partito* es aquél cuyo conjunto de vértices puede ser particionado en k subconjuntos tales que cada vértice puede ser adyacente únicamente a vértices fuera del subconjunto al que pertenece.
En particular, un grafo *bipartito* es aquél cuyos vértices pueden ser particionados en dos subconjuntos X y Y tales que para todo $x \in X$, se cumple que, si $d(x) \neq 0$, x es adyacente a algún vértice $y \in Y$. Análogamente, $\forall y \in Y$ existe algún $x \in X$ tal que y es adyacente a x; si un grafo es bipartito simple y todos los vértices en X son adyacentes a todos los vértices de Y y viceversa, el grafo es un *grafo bipartito*

[1] Para un conjunto C, se denota como $|C|$ a la cardinalidad de C.
[2] Dos grafos G y H son isomorfos si existen biyecciones $\theta : V_G \to V_H$ y $\phi : E_G \to E_H$ tales que $\psi_G(e) = (u, v)$ si y sólo si $\psi_H(\phi(e)) = (\theta(u), \theta(v))$. El capítulo 2 de [20] se aborda el concepto con mayor detalle.

completo. Si $|X| = m$ y $|Y| = n$, el grafo G se denota como $K_{m,n}$.

Ejemplo 99. La primera imagen muestra un grafo bipartito (cuyas particiones son $\{v_1, v_3, v_4\}$ y $\{v_2, v_5\}$); en la segunda vemos un grafo bipartito completo (todos los elementos de la partición $\{v_1, v_2, v_3\}$ están conectados a todos los elementos de la partición $\{v_4, v_5, v_6\}$ y viceversa).





Definición 9. Sea un grafo $G = (V_G, E_G, \psi_G)$. Un grafo $H = (V_H, E_H, \psi_H)$ es *subgrafo* de G si $V_H \subseteq V_G$, $E_H \subseteq E_G$ y ψ_H es la restricción de ψ_G para E_H.

Definición 10. La *matriz de incidencia* de un grafo G es una matriz binaria de $|V| \times |E|$ en la que cada elemento $[m_{ij}]$ cumple lo siguiente:

$$[m_{ij}] = \begin{cases} 0 & \text{si la arista } e_j \text{ no es incidente en el vértice } v_i. \\ 1 & \text{si la arista } e_j \text{ es incidente en el vértice } v_i. \end{cases}$$

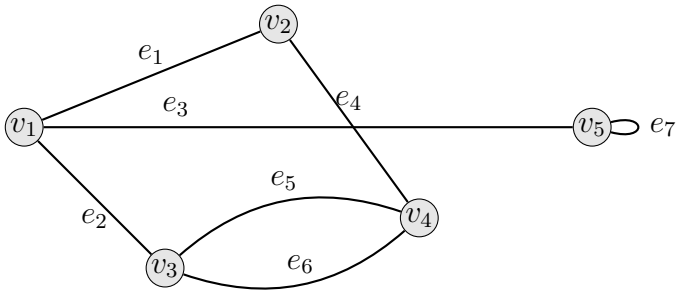
Definida de la forma anterior, teniendo un grafo simple, sucederá que la suma de cada una de las columnas de la matriz siempre será 2, lo que quiere decir que cualquier arista es incidente a exactamente dos vértices.
Cuando se trabaja con una gráfica no simple, en algunas ocasiones se opta por eliminar el elemento binario de la matriz para ser más específico en el número de aristas que inciden en algún vértice.

Definición 11. La *matriz de adyacencia* de un grafo G es una matriz binaria de $|V| \times |V|$ en la que cada elemento $[m_{ij}]$ cumple lo siguiente:

$$[m_{ij}] = \begin{cases} 0 & \text{si el vértice } v_i \text{ no es adyacente al vértice } v_j. \\ 1 & \text{si el vértice } v_i \text{ es adyacente al vértice } v_j. \end{cases}$$

Al igual que la matriz de incidencia, ésta es una representación computacional muy usual para un grafo, y cuando se trabaja con una gráfica no simple, es posible, en lugar de 1, especificar el número de veces en que un vértice es adyacente a otro.

Ejemplo 100. Obtener la matriz de incidencia y de adyacencia en su versión binaria del siguiente grafo.



Solución.

Matriz de incidencia:

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{ccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 \end{array} \\ \left[\begin{array}{ccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

Matriz de adyacencia:

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \left[\begin{array}{ccccc} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

Nota que la matriz de adyacencia resulta ser simétrica, sin embargo, este hecho puede cambiar si se trabaja con una gráfica dirigida.

Ejemplo 101. Obtener la matriz de incidencia y de adyacencia en su versión no binaria del grafo mostrado en el ejemplo anterior.

Solución. Ambas matrices resultarán muy similares a las de la solución del ejemplo anterior, pero observaremos algunos cambios: El elemento $(5, 7)$ de la matriz de incidencia adopta el valor de 2 haciendo referencia a que la arista e_7 incide dos veces en el vértice v_5; por otro lado, los elementos $(4, 3)$ y $(3, 4)$ de la matriz de adyacencia toman el valor de 2, implicando que son adyacentes entre sí dos ocasiones distintas (una a través de e_5 y otra a través de v_6).

Matriz de incidencia:

	e_1	e_2	e_3	e_4	e_5	e_6	e_7
v_1	1	1	1	0	0	0	0
v_2	1	0	0	1	0	0	0
v_3	0	1	0	0	1	1	0
v_4	0	0	0	1	1	1	0
v_5	0	0	1	0	0	0	**2**

Matriz de adyacencia:

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	0	1
v_2	1	0	0	1	0
v_3	1	0	0	**2**	0
v_4	0	1	**2**	0	0
v_5	1	0	0	0	1

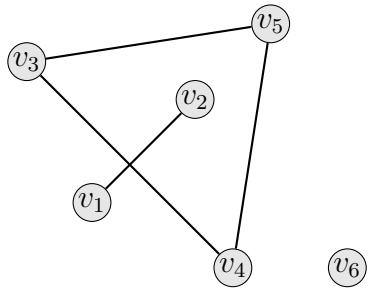
Definición 12. Un *camino* o *trayectoria* es una sucesión de vértices y aristas $v_0, e_1, v_1, \ldots, e_k, v_k$ no necesariamente distintos, en donde, para cada $e_i, 1 \leq i \leq k$, se cumple que e_i es incidente en los vértices v_{i-1} y v_i.

Definición 13. Un *ciclo* o *camino cerrado* es un camino en el que $v_0 = v_k$.
Cuando un grafo no contiene ciclos es llamado *acíclico*.

Definición 14. Un grafo no dirigido es *conectado* o *conexo* si para todo par de vértices u y v existe al menos un camino para llegar de uno a otro.

Definición 15. Sea $G = (V, E, \psi)$ un grafo. Un *componente* C de G es un subgrafo conectado en el cual, para todo vértice $u \in C$, se cumple que no existe un camino de u a v para $v \notin C$.

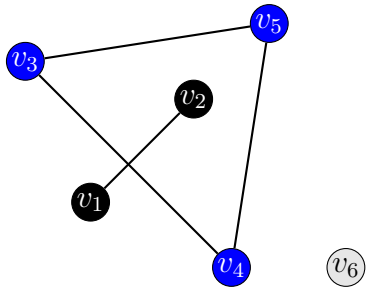
Ejemplo 102. Determina los componentes del siguiente grafo.



Solución. Ya que no existe arista alguna incidente a v_6, podemos concluir que v_6 constituye un componente; luego, existe una arista que une a v_1 y v_2, que no son adyacentes a más vértices, por lo que aquí tenemos otro componente; finalmente, v_3, v_4 y v_5 conforman el último componente.

Es probable que el ver a los componentes $\{v_1, v_2\}$ y $\{v_3, v_4, v_5\}$ de la manera en que están ilustrados hayas caído en la confusión de que eran uno mismo, sin embargo, debes tener claro que estos únicamente están traslapados, pero no hay ninguna arista que funja como la unión de ellos, por lo cual son dos componentes distintos.

Si coloreamos los vértices de cada componente de un color distinto, el grafo del ejercicio se vería como sigue:



Definición 16. Un vértice v es *vecino* de un vértice u si existe la arista uv. El conjunto de los vecinos de v se denota como $N(v)$.

Definición 17. Sea G un grafo. El *grado* de un vértice v es la cardinalidad del conjunto de vértices vecinos de v. La simbología para el grado de v suele ser $d(v)$.

Teorema 15. Sea $G = (V, E, \psi)$ un grafo simple. Se cumple que:

$$\sum_{v \in V} d(v) = 2|E|$$

Es decir, la suma del grado de todos sus vértices será lo mismo que dos veces la cantidad de sus aristas.

Demostración. Sea $M = [m_{ij}]$ la matriz de incidencia del grafo G. Entonces, para cada vértice $v_i \in V$, $d(v_i) = \sum_{j=1}^{|E|} m_{ij}$.

Por lo anterior, la suma del grado de todos los vértices se puede escribir como sigue.

$$\sum_{v \in V} d(v) = \sum_{i=1}^{|V|} \sum_{j=1}^{|E|} m_{ij} = \sum_{j=1}^{|E|} \sum_{i=1}^{|V|} m_{ij}$$

En la definición de *matriz de incidencia* vimos que, para un grafo simple, la suma de cada columna en su matriz de incidencia será 2, así, para cada $i = 1, 2, \ldots, |V|$,

$$\sum_{i=1}^{|V|} m_{ij} = 2$$

Finalmente, y considerando todas las aristas (columnas en la matriz de incidencia):

$$\sum_{j=1}^{|E|} \sum_{i=1}^{|V|} m_{ij} = 2|E|$$

□

Teorema 16. Para toda grafo simple, se cumple que el número de vértices de grado impar es par.

Demostración. Sea $G = (V, E, \psi)$ un grafo simple, y sea $\{V_{par}, V_{impar}\}$ una partición sobre G con respecto a la paridad del grado de los elementos de V. Se sigue entonces que:

$$\sum_{v \in V} d(v) = \sum_{v \in V_{par}} d(v) + \sum_{v \in V_{impar}} d(v)$$

$$\Longrightarrow \sum_{v \in V_{impar}} d(v) = \sum_{v \in V} d(v) - \sum_{v \in V_{par}} d(v)$$

Es claro que $\sum_{v \in V_{par}} d(v)$ es un número par sin importar los sumandos involucrados; además, por el teorema anterior, se sabe que $\sum_{v \in V} d(v) = 2|E|$, es decir, un número par. Por lo tanto, la resta de ellos, igual a $\sum_{v \in V} d(v)$, es también un número par.

□

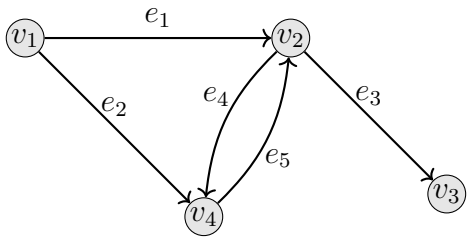
3.2. Grafos Dirigidos

En ocasiones, para representar correctamente un conjunto de datos no basta con los grafos descritos anteriormente. Para resolver algunos problemas, es necesario agregar una propiedad a las aristas: *dirección*. Esto con el fin de especificar si la relación establecida entre dos vértices u y v es de u a v, de v a u, o bien, si están presentes ambas relaciones.

Considerar la dirección en una arista entre dos vértices implicará especificar si es posible llegar de u a v y/o de v a u, cambiando con ello la representación de la gráfica en su matriz de adyacencia.

Definición 18. Un *grafo dirigido* o *digrafo* D se define como una tripleta (V, E, ψ), en la que V es un conjunto no vacío de vértices, E es un conjunto de aristas y ψ es una función de incidencia que asocia a cada arista en E con un par ordenado de vértices no necesariamente distintos en V.
Si el par ordenado de vértices que corresponde a $e_i \in E$ es (u, v), u es llamado *vértice de origen* y v es llamado *vértice de llegada*.

Ejemplo 103. Identifica los elementos de la tripleta (V, E, ψ) que define al grafo dirigido mostrado a continuación.



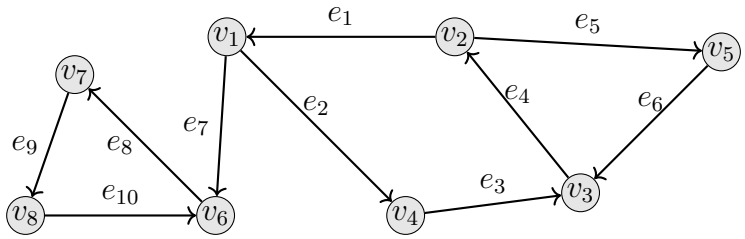
Solución.

$$
\begin{aligned}
V &= \{v_1, v_2, v_3, v_4\} \\
E &= \{e_1, e_2, e_3, e_4, e_5\} \\
\psi(e_1) &= (v_1, v_2), \\
\psi(e_2) &= (v_1, v_4), \\
\psi(e_3) &= (v_2, v_3), \\
\psi(e_4) &= (v_2, v_4), \\
\psi(e_5) &= (v_4, v_2)
\end{aligned}
$$

El elemento de la dirección en las aristas de un grafo acíclico dirigido da lugar a un ordenamiento que depende de los vértices de origen y los vértices de llegada de cada arista, y que es conocido como *orden topológico*.

Definición 19. Se dice que un grafo dirigido $D = (V, E, \psi)$ tiene *componentes fuertemente conexos* si V puede ser descompuesto en subconjuntos, cada uno con la propiedad de que todos sus vértices son mutuamente alcanzables.

Ejemplo 104. Identifica los componentes fuertemente conexos del siguiente digrafo.



Solución. En nuestro digrafo existen 2 componentes fuertemente conexos:

$$C_1 = \{v_1, v_2, v_3, v_4, v_5\}$$
$$C_2 = \{v_6, v_7, v_8\}$$

Para entender por qué, primero comprobemos que todos los vértices del componente C_1 sean mutuamente accesibles, para lo cual, mostraremos las trayectorias entre cada par de vértices.

$$(v_1, v_2) : v_1 - e_2 - v_4 - e_3 - v_3 - e_4 - v_2$$
$$(v_1, v_3) : v_1 - e_2 - v_4 - e_3 - v_3$$
$$(v_1, v_4) : v_1 - e_2 - v_4$$
$$(v_1, v_5) : v_1 - e_2 - v_4 - e_3 - v_3 - e_4 - v_2 - e_5 - v_5$$

$$(v_2, v_1) : v_2 - e_1 - v_1$$
$$(v_2, v_3) : v_2 - e_5 - v_5 - e_6 - v_3$$
$$(v_2, v_4) : v_2 - e_1 - v_1 - e_2 - v_4$$
$$(v_2, v_5) : v_2 - e_5 - v_5$$

$$(v_3, v_1) : v_3 - e_4 - v_2 - e_1 - v_1$$
$$(v_3, v_2) : v_3 - e_4 - v_2$$
$$(v_3, v_4) : v_3 - e_4 - v_2 - e_1 - v_1 - e_2 - v_4$$
$$(v_3, v_5) : v_3 - e_4 - v_2 - e_5 - v_5$$

$$(v_4, v_1) : v_4 - e_3 - v_3 - e_4 - v_2 - e_1 - v_1$$
$$(v_4, v_2) : v_4 - e_3 - v_3 - e_4 - v_2$$
$$(v_4, v_3) : v_4 - e_3 - v_3$$

$$(v_4, v_5) : v_4 - e_3 - v_3 - e_4 - v_2 - e_5 - v_5$$

$$(v_5, v_1) : v_5 - e_6 - v_3 - e_4 - v_2 - e_1 - v_1$$
$$(v_5, v_2) : v_5 - e_6 - v_3 - e_4 - v_2$$
$$(v_5, v_3) : v_5 - e_6 - v_3$$
$$(v_5, v_4) : v_5 - e_6 - v_3 - e_4 - v_2 - e_1 - v_1 - e_2 - v_4$$

Nota que para algunos pares de vértices había más de una manera de llegar de uno a otro. Eso, en realidad, es algo sin importancia en este tema.

Volviendo a nuestra verificación, efectivamente, todos los vértices son mutuamente alcanzables, de lo que deducimos que el subconjunto $\{v_1, v_2, v_3, v_4, v_5\}$ forma un componente fuertemente conexo del digrafo dado.

De la misma forma tratamos al subconjunto $\{v_6, v_7, v_8\}$:

$$(v_6, v_7) : v_6 - e_8 - v_7$$
$$(v_6, v_8) : v_6 - e_8 - v_7 - e_9 - v_8$$

$$(v_7, v_6) : v_7 - e_9 - v_8 - e_{10} - v_6$$
$$(v_7, v_8) : v_7 - e_9 - v_8$$

$$(v_8, v_6) : v_8 - e_{10} - v_6$$
$$(v_8, v_7) : v_8 - e_{10} - v_6 - e_8 - v_7$$

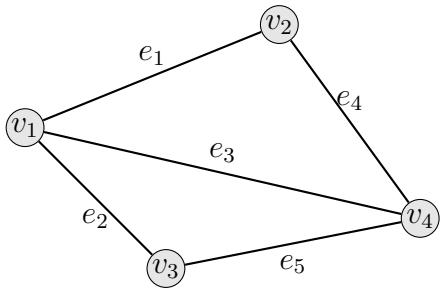
3.3. Representación de un Grafo

En el área de la programación, existen varias formas para representar un grafo (algunas de ellas ya las hemos abordado). Para cada una, existen diversas ventajas

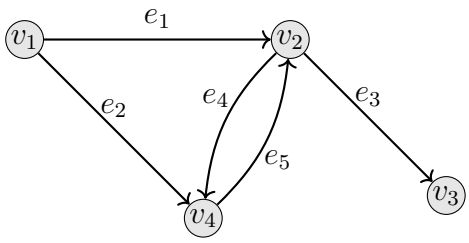
y desventajas; cada una ofrece utilidades distintas, y de acuerdo a ellas se elige el método adecuado para cada problema.

Considera los siguientes grafos.

Grafo A



Grafo B



Es importante mencionar que, antes de manejar los datos, se requiere tener conocimiento de si se está trabajando con un grafo dirigido o con uno no dirigido, pues de eso dependerá la manera en que se almacene la gráfica.

3.3.1. Lista de Aristas

Es usual que para representar un grafo $G = (V, E, \psi)$ se haga uso de una *lista de aristas*. Ésta se define como un arreglo de pares en el que cada elemento representa una arista $v_i v_j$, y donde cada par contiene a los vértices v_i y v_j.

Normalmente, la lista de aristas no se utiliza para realizar algoritmos sobre un grafo, sino como descripción del mismo. La entrada de los problemas de grafos suelen ser las aristas justamente con este formato, por lo que parece natural hacer uso de esta estructura para su lectura.

Ejemplo 105. Para el grafo A mostrado al principio de esta sección, construye la lista de aristas que le corresponde y el fragmento de programa requerido para leer y guardar los datos de entrada.

Solución. Comencemos por establecer la lista de aristas.

$$(v_1, v_2)$$
$$(v_1, v_3)$$
$$(v_1, v_4)$$
$$(v_2, v_4)$$
$$(v_3, v_4)$$

Recordemos que estamos tratando con un grafo no dirigido, lo que implica que para cualquier par de vértices v_i, v_j para el que exista la arista v_iv_j, es posible llegar de v_i a v_j y viceversa. En el caso de la lista de aristas, resulta impráctico guardar los pares (v_i, v_j) y (v_j, v_i), por lo que, para resolver un problema a partir de una lista de aristas, únicamente es necesario tener en cuenta el tipo de grafo con que se está trabajando.

Por otro lado, el fragmento de programa para la lectura y almacenamiento de la información es:

```
1  void solucion()
2  {
3      int aristas, v1, v2, i;
4      vector<pair<int, int>> lista_aristas;
5
6      cin>>aristas;
7      for (i = 0; i < aristas; i = i + 1)
8      {
9          cin>>v1>>v2;
10         lista_aristas.push_back({v1, v2});
11     }
12 }
```

Ejemplo 106. Para el grafo B, mostrado al principio de esta sección, construye la lista de aristas que le corresponde y el fragmento de programa requerido para leer y guardar los datos de entrada.

Solución. La lista de aristas del grafo B es:

$$(v_1, v_2)$$
$$(v_1, v_4)$$
$$(v_2, v_3)$$
$$(v_2, v_4)$$
$$(v_4, v_2)$$

En cuanto al programa:

```
1  void solucion()
2  {
3      int aristas, v1, v2, i;
4      vector<pair<int, int>> lista_aristas;
5
6      cin>>aristas;
7      for (i = 0; i < aristas; i = i + 1)
8      {
9          cin>>v1>>v2;
10         lista_aristas.push_back({v1, v2});
11     }
12 }
```

3.3.2. Matriz de Adyacencia

Como ya fue explicado en la sección de *Conceptos Básicos*, la matriz de adyacencia de un grafo $G = (V, E, \psi)$ describe la existencia o no existencia de todas las aristas resultantes del conjunto $\binom{[V]}{1} \cup \binom{[V]}{2}$.

Normalmente se ocupa la matriz de adyacencia en grafos con no más de 1000 vértices, pues requiere un espacio de almacenamiento equivalente al cuadrado de la cantidad de vértices.

Como ventaja, podemos mencionar que es posible saber en tiempo constante si dos vértices son vecinos haciendo una consulta en cualquier posición de la matriz, y como desventaja, encontramos el hecho de que recorrer los vecinos de un vértice tiene una

complejidad lineal, pues es necesario recorrer toda la línea (ya sea columna o renglón) a la que el vértice pertenece. Podemos añadir, además, que cuando un grafo tiene pocas aristas, éste resultará en una matriz de adyacencia poco densa, provocando un gasto de memoria innecesario.

Ejemplo 107. Obtén la matriz de adyacencia correspondiente al grafo A. Escribe el fragmento de programa requerido para la lectura y almacenamiento de los datos.

Solución.

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

```
//El tamaño dependerá de los vértices de la matriz.
//Siendo variable global, por default está llena de 0's.
int matriz_adyacencia[1001][1001];

void solucion()
{
    int aristas, v1, v2, i;

    cin>>aristas;
    for (i = 0; i < aristas; i = i + 1)
    {
        cin>>v1>>v2;
        //La relación es bidireccional.
        matriz_adyacencia[v1][v2] = 1;
        matriz_adyacencia[v2][v1] = 1;
    }
}
```

Ejemplo 108. Obtén la matriz de adyacencia correspondiente al grafo B. Escribe el fragmento de programa requerido para la lectura y almacenamiento de los datos.

Solución.

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

```
//El tamaño dependerá de los vértices de la matriz.
//Siendo variable global, por default está llena de 0's.
int matriz_adyacencia[1001][1001];

void solucion()
{
    int aristas, v1, v2, i;

    cin>>aristas;
    for (i = 0; i < aristas; i = i + 1)
    {
        cin>>v1>>v2;
        //La relación es unidireccional.
        matriz_adyacencia[v1][v2] = 1;
    }
}
```

Como ya se señaló antes, mientras la matriz de adyacencia de un grafo que no es dirigido es simétrico, la de un grafo dirigido no necesariamente lo es, pues en el caso de este último, la existencia de la arista (v_i, v_j) no asegura la existencia de la (v_j, v_i).

3.3.3. Lista de Adyacencia

Esta estructura es un arreglo de listas ligadas, en donde, para algún vértice v_i, $lista[i]$ contiene al vértice v_j si y sólo si la arista $v_i v_j$ existe. Así, la longitud del arreglo es $|V|$, en el que cada elemento es una lista que contiene a los vecinos del vértice v_i.

La lista de adyacencia es la estructura de datos más utilizada para la representación

y almacenamiento de grafos, pues permite recorrer los vecinos de un determinado vértice v_i más rápido que el resto de los métodos al pasar únicamente por sus vértices adyacentes; además, la memoria que requiere es igual al número de aristas en la gráfica, o el doble si es no dirigida. En contraste, si lo que se requiere es averiguar si dos vértices v_i, v_j son vecinos, la lista de adyacencia está en desventaja contra la matriz de adyacencia, ya que habrá que recorrer todos vecinos, ya sea de v_i o de v_j.

Ejemplo 109. Construye la lista de adyacencia del grafo A y escribe el fragmento de programa necesario para su lectura y almacenamiento.

Solución.

$$\begin{aligned} v_1 &\to \{v_2, v_3, v_4\} \\ v_2 &\to \{v_1, v_4\} \\ v_3 &\to \{v_1, v_4\} \\ v_4 &\to \{v_1, v_2, v_3\} \end{aligned}$$

```
vector<vector<int>> lista_adyacencia;

void solucion()
{
    int vertices, aristas, i, v1, v2;

    cin>>vertices>>aristas;
    lista_adyacencia.resize(vertices);

    for (i = 0; i < aristas; i = i + 1)
    {
        cin>>v1>>v2;
        //Las aristas son bidireccionales.
        lista_adyacencia[v1].push_back(v2);
        lista_adyacencia[v2].push_back(v1);
    }
}
```

Ejemplo 110. Construye la lista de adyacencia del grafo B y escribe el fragmento de programa necesario para su lectura y almacenamiento.

Solución.

$$v_1 \rightarrow \{v_2, v_4\}$$
$$v_2 \rightarrow \{v_3, v_4\}$$
$$v_3 \rightarrow \emptyset$$
$$v_4 \rightarrow \{v_2\}$$

```
vector<vector<int>> lista_adyacencia;

void solucion()
{
    int vertices, aristas, i, v1, v2;

    cin>>vertices>>aristas;
    lista_adyacencia.resize(vertices);

    for (i = 0; i < aristas; i = i + 1)
    {
        cin>>v1>>v2;
        lista_adyacencia[v1].push_back(v2);
    }
}
```

3.3.4. Matriz de Incidencia

En realidad, esta manera de representar un grafo es la menos popular en el ámbito de la programación competitiva, pues por lo general, los algoritmos usan las relaciones entre vértices y no entre vértices y aristas, además, se requiere demasiada memoria para almacenar las incidencias de cada arista. De cualquier forma, la estudiaremos en caso de requerirse, y dicho sea de paso, puede ser una opción viable para tratar con hipergrafos[3].

A diferencia de las estructuras vistas antes, la matriz de incidencia se define de distinta manera para grafos dirigidos y no dirigidos. Retomando la definición antes vista de matriz de incidencia, este arreglo tendrá dimensiones $|V| \times |E|$. En el caso de los grafos no dirigidos, el elemento en la posición (i, j) será 1 si la arista e_j incide en el

[3] Un hipergrafo es un grafo en el que cada una de sus aristas pueden unir a más de dos vértices.

vértice v_i, de lo contrario tomará el valor de 0. Por otro lado, la matriz de incidencia de un grafo dirigido definirá el valor de sus posiciones de la siguiente manera:

$$(i,j) = \begin{cases} 1 & \text{si el vértice } v_i \text{ es el vértice de origen de la arista } e_j. \\ -1 & \text{si el vértice } v_i \text{ es el vértice de llegada de la arista } e_j. \\ 0 & \text{en cualquier otro caso.} \end{cases}$$

Ejemplo 111. ¿Cuál es la matriz de incidencia del grafo A? ¿Cuál es el fragmento de programa requerido para llenar dicha matriz?

Solución.

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

```
1  //Suponiendo que no haya más de 10000 aristas y 1000 vértices.
2  int matriz_incidencia[1001][10001];
3
4  void solucion()
5  {
6      int aristas, j, v1, v2;
7
8      cin>>aristas;
9      for (j = 1; j <= aristas; j = j + 1)
10     {
11         cin>>v1>>v2;
12         matrizIncidencia[v1][j] = 1;
13         matrizIncidencia[v2][j] = 1;
14     }
15 }
```

Ejemplo 112. ¿Cuál es la matriz de incidencia del grafo B? ¿Cuál es el fragmento de programa requerido para llenar dicha matriz?

Solución.

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 1 & -1 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 1 \end{bmatrix} \end{array}$$

```
1  //Suponiendo que no haya más de 10000 aristas y 1000 vértices.
2  int matriz_incidencia[1001][10001];
3
4  void solucion()
5  {
6      int aristas, j, v1, v2;
7
8      cin>>aristas;
9      for (j = 1; j <= aristas; j = j + 1)
10     {
11         cin>>v1>>v2;
12         matrizIncidencia[v1][j] = 1;
13         matrizIncidencia[v2][j] = -1;
14     }
15 }
```

3.4. Grafos con Pesos

El *peso* de una arista uv es una medida que se le asigna a la misma con la intención de representar el costo, tiempo, distancia u otra medida, que toma llegar desde el vértice u al vértice v; el peso de la arista uv suele denotarse como $w(u, v)$. Así, se puede definir al peso como una función $w : V \times V \to \mathbb{R}$, y a un grafo con pesos como una 4-tupla (V, E, ψ, w).

Dicho esto, abordemos algunos resultados interesantes aplicables a grafos con pesos.

3.4.1. Ruta de Menor Peso

En muchas situaciones puede necesitarse la obtención de una ruta cuyo peso sea el menor entre todas las trayectorias posibles, pues claramente ése será el camino que menos recursos requiera. Saber un algoritmo que resuelva este problema representa una gran ventaja a la hora de resolver interrogantes relativas a teoría de grafos, pues de esta forma estaremos evitando las costosas búsquedas completas.

En esta sección abordaremos tres algoritmos que encuentran la ruta de menor peso en una gráfica, pero hay que enfatizar que todos ellos requieren condiciones distintas y se usan en distintos contextos, mismos que aclararemos en lo sucesivo.

Algoritmo de Dijkstra

Este algoritmo es útil para encontrar la ruta de menor peso entre un vértice v y todos los vértices para los que existe una ruta a partir de dicho vértice en un grafo dirigido, es decir, encontrará la ruta menos costosa para ir de un vértice elegido, a cada vértice en el grafo para que el haya una trayectoria a partir de él. Para encontrar esta ruta, el algoritmo de Dijkstra se apoya en el uso de *etiquetas* numéricas (usualmente denotadas por $l(v)$) asignadas a los vértices, que se modifican en caso de que se encuentre una mejor manera de llegar a ellos. A continuación, describiremos el funcionamiento del algoritmo.

Al inicio, se elige un vértice *fuente*, al que denotaremos como s (a partir del que se buscarán las rutas óptimas) y se le da un valor de 0 a su etiqueta; al resto de los vértices se les asigna una etiqueta con un valor muy alto (en la teoría, se les asigna el valor de ∞). Paso siguiente, se ubican los vecinos de s y se sustituye su etiqueta por la suma de $l(s)$ más el peso de la arista que los comunica; con base en lo anterior elegimos al siguiente vértice a analizar, y será el que tenga la menor etiqueta, además, recordamos a s como su predecesor; por comodidad, nos referiremos al vértice siguiente (el que se eligió como el propietario de la menor etiqueta) como u_1. Luego, fijamos nuestra atención en los vértices de llegada de las aristas cuyo vértice de origen es u_1, y aquí debemos tomar una decisión: si la etiqueta de esos vértices

es mayor que la suma de la etiqueta de u_1 más el peso de la arista que los comunica, entonces lo sustituimos por el valor de la mencionada adición, de lo contrario, el valor de la etiqueta en los vértices analizados queda igual; nuevamente, es necesario recordar a su predecesor (u_1) para reconstruir la ruta. Este procedimiento se repetirá hasta que se haya pasado por todos los vértices de la gráfica. Nota que, de hecho, éste fue el procedimiento que se aplicó en la primera iteración, pero omitimos la decisión debido a que todas las etiquetas estaban en ∞.

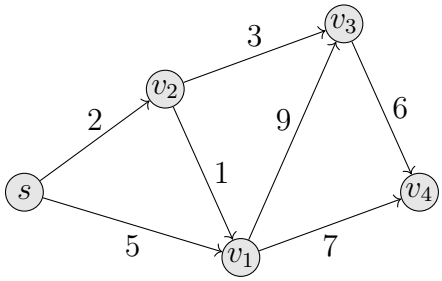
Lo que hemos descrito anteriormente se puede representar con el siguiente algoritmo.

Algoritmo Dijkstra

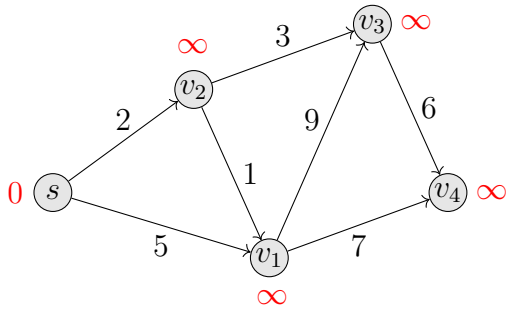
1: Procesado $P \leftarrow \emptyset$
2: **para cada** $u \in V$ **hacer**
3: $\quad l(u) \leftarrow \infty$
4: **fin para**
5: $l(s) \leftarrow 0$
6: $pred(s) \leftarrow null$
7: **mientras** $P \neq V$ **hacer**
8: $\quad u \leftarrow v \in V - P$ tal que $l(u) = \text{mín}\{l(v) | v \in V - P\}$
9: $\quad P \leftarrow P \cup \{u\}$
10: $\quad$ **para todo** $v \in N(u) \cap V - P$ tal que $l(v) > l(u) + w(u, v)$ **hacer**
11: $\quad\quad l(v) \leftarrow l(u) + w(u, v)$
12: $\quad\quad pred(v) \leftarrow u$
13: $\quad$ **fin para**
14: **fin mientras**

Para comprender mejor el algoritmo, veamos su aplicación, paso a paso, en una gráfica dirigida.

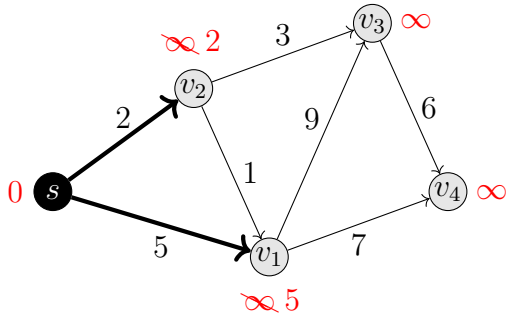
Ejemplo 113. Aplica el algoritmo de Dijkstra a la gráfica mostrada a continuación.



Solución. Lo primero es asignar las etiquetas a cada vértice, mismas que, como apoyo, están escritas en rojo.



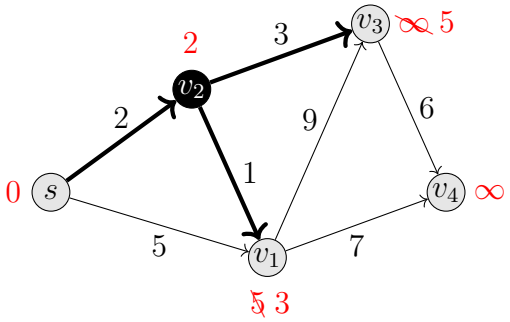
Comenzamos el análisis por los vecinos de s, sustituyendo sus etiquetas por $l(s)$ (=0) más el peso de las aristas que las comunican con la fuente:



De esta forma, decimos que el predecesor de v_1 y v_2 es s, y que, hasta este punto, la ruta más corta para llegar a ellos es pasar por las aristas sv_1 y sv_2, respectivamente. Nuestros datos se pueden resumir como muestra la siguiente tabla.

Iteración	P	Predecesores
1	$\{s\}$	v_1: s, v_2: s

Ya que la etiqueta de v_2 es menor que la de v_1, la siguiente iteración se basará en analizar a los vecinos de v_2.

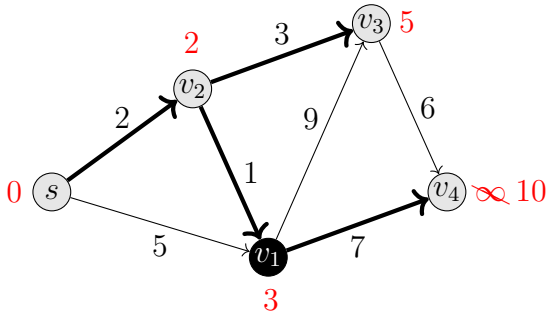


En esta iteración se modificaron las etiquetas de v_1 y v_3. Nota que, aunque $l(v_1)$ ya era menor que ∞, fue necesario cambiarla, pues $l(v_2)+w(v_2, v_1) = 2 + 1$ es menor que 5, su etiqueta anterior; con esto, el predecesor de v_1 se cambia por v_2, y el de v_3 se fija en v_2. Por tanto podemos ver que, hasta el momento, la ruta menos costosa de s a v_1 es $s - v_2 - v_1$ (y no directamente ir de s a v_1), y la de s a v_3 es $s - v_2 - v_3$.

Iteración	P	Predecesores
1	$\{s\}$	v_1: s, v_2: s
2	$\{s, v_2\}$	v_1: v_2, v_2: s, v_3: v_2

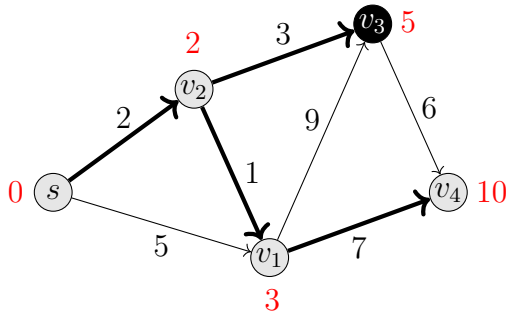
En la tercera iteración, debemos analizar los vecinos del vértice vecino a v_2 con la menor etiqueta, esto es v_1.

Aquí, aunque v_3 es analizado nuevamente por ser vecino de v_1, su predecesor no cambia pues 5 es menor que $9 + 3$:



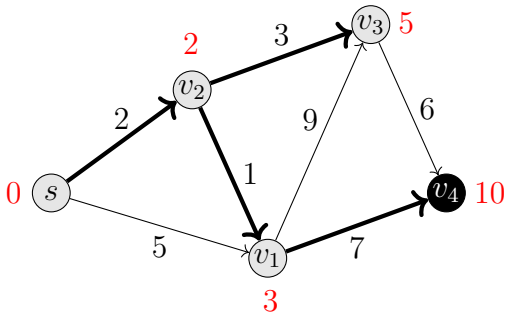
Iteración	P	Predecesores
1	$\{s\}$	v_1: s, v_2: s
2	$\{s, v_2\}$	v_1: v_2, v_2: s, v_3: v_2
3	$\{s, v_2, v_1\}$	v_1: v_2, v_2: s, v_3: v_2, v_4: v_1

En la cuarta iteración, analizando los vecinos de v_3, no realizamos ninguna modificación, pues su único vecino, v_4, tiene una etiqueta menor que $l(v_3) + w(v_3, v_4)$:



Iteración	P	Predecesores
1	$\{s\}$	v_1: s, v_2: s
2	$\{s, v_2\}$	v_1: v_2, v_2: s, v_3: v_2
3	$\{s, v_2, v_1\}$	v_1: v_2, v_2: s, v_3: v_2, v_4: v_1
4	$\{s, v_2, v_1, v_3\}$	v_1: v_2, v_2: s, v_3: v_2, v_4: v_1

Finalmente:



Iteración	P	Predecesores
1	$\{s\}$	v_1: s, v_2: s
2	$\{s, v_2\}$	v_1: v_2, v_2: s, v_3: v_2
3	$\{s, v_2, v_1\}$	v_1: v_2, v_2: s, v_3: v_2, v_4: v_1
4	$\{s, v_2, v_1, v_3\}$	v_1: v_2, v_2: s, v_3: v_2, v_4: v_1
5	$\{s, v_2, v_1, v_3, v_4\}$	v_1: v_2, v_2: s, v_3: v_2, v_4: v_1

En este punto, el conjunto P ya tiene a todos los vértices de la gráfica, por lo que el algoritmo ha terminado.

Ahora bien, interpretemos lo que el algoritmo ha arrojado: Gracias al almacenamiento de los predecesores de cada vértice, es posible obtener la ruta óptima a v_1, v_2, v_3 y v_4, así como el costo que toma llegar a cada uno al revisar las etiquetas.

Ejemplo 114. Dada una gráfica dirigida conexa con pesos, encuentra la ruta de menor peso para ir de un vértice a otro.

Solución. En este ejemplo nos ocuparemos de programar el algoritmo de Dijkstra, recordando que la salida será una sucesión de vértices, y suponiendo que ya hemos almacenado el grafo dado en *lista_adyacencia*.

```
1  #define INF 100000000
2
3  vector<int> Dijkstra_ruta(vector<vector<pair<int, int>>>
   ↪  lista_adyacencia, int s, int t)
4  {
5      int n = lista_adyacencia.size(), procesados = 0;
6      vector<int> pred(n);
```

```
7
8       bitset<10000> procesado;
9
10      vector<int> l(n, INF); //Asignación de etiquetas.
11      priority_queue<pair<int, int>, vector<pair<int, int>>,
12                     greater<pair<int, int>>> cola_prioridad;
13
14      l[s] = 0;
15      pred[s] = 0; //null
16      cola_prioridad.push({l[s], s});
17
18      n = n - 1;  //Eliminar el vértice 0, no existe.
19
20      while (procesados != n)
21      {
22          int u = cola_prioridad.top().second;
23          cola_prioridad.pop();
24          procesado[u] = true;
25          procesados = procesados + 1;
26
27          for (auto &par: lista_adyacencia[u])
28          {
29              int v = par.first;
30              if (procesado[v])
31                  continue;
32
33              int w = par.second;
34              if (l[v] > l[u] + w)
35              {
36                  l[v] = l[u] + w;
37                  pred[v] = u;
38              }
39          }
40      }
41
42      //Para reconstruir la ruta.
43      vector<int> sucesion;
44      while (t)
45      {
46          sucesion.push_back(t);
47          t = pred[t];
48      }
49      reverse(sucesion.begin(), sucesion.end());
50
51      return sucesion;
52  }
```

```
53
54 void sucesion()
55 {
56     int v1, v2;
57     cin>>v1>>v2;
58     auto sucesion = Dijkstra(lista_adyacencia, v1, v2);
59
60     for (int vertice: sucesion)
61         cout<<vertice<<" ";
62 }
```

Ejemplo 115. Imagina una ciudad en la que existen exactamente k edificios por los que pasan carreteras para las que conoces el tiempo que tardarías en recorrerlas. Imagina también que en dicha ciudad hay un total de b estaciones de bomberos. Así pues, se intenta implantar un sistema en el que todas las llamadas al cuerpo de bomberos sean atendidas en el menor tiempo posible, y dado que los habitantes son personas muy optimistas, se manejan bajo la premisa de que una estación se deberá hacer cargo de únicamente un evento a la vez.

Para cierto edificio dado (distinto a las estaciones de bomberos), determina cuál es el tiempo que tomaría recorrer el camino entre él y la estación de bomberos más cercana.

> **Solución.** Este problema tiene dos variaciones con respecto al ejemplo anterior: la primera es que en este caso tenemos más de un origen: todos los edificios de bomberos, y la segunda es que debemos encontrar el peso de la ruta óptima.

```
1  #define INF 100000000
2
3  vector<int> Dijkstra_peso(vector<vector<pair<int, int>>>
   ↪  lista_adyacencia, vector<int> S)
4  {
5      int n = lista_adyacencia.size(), procesados = 0;
6
7      bitset<10000> procesado;
8
9      vector<int> l(n, INF); //Asignación de etiquetas.
10     priority_queue<pair<int, int>, vector<pair<int, int>>,
11                    greater<pair<int, int>>> cola_prioridad;
```

```
12
13      for (auto s: S)
14      {
15          l[s] = 0;
16          cola_prioridad.push({l[s], s});
17      }
18
19      n = n - 1;  //Eliminar el vértice 0, no existe.
20
21      while (procesados != n)
22      {
23          int u = cola_prioridad.top().second;
24          cola_prioridad.pop();
25          procesado[u] = true;
26          procesados = procesados + 1;
27
28          for (auto &par: lista_adyacencia[u])
29          {
30              int v = par.first;
31              if (procesado[v])
32                  continue;
33
34              int w = par.second;
35              if (l[v] > l[u] + w)
36                  l[v] = l[u] + w;
37          }
38      }
39
40      return l;
41  }
42
43  void solucion()
44  {
45      vector<int> bomberos, edificios;
46      int n, b, i, n_edif, e;
47
48      cin>>n;
49      for (i = 0; i < n; i = i + 1)
50      {
51          cin>>b;
52          bomberos.push_back(b);
53      }
54      for (i = 0; i < n_edif; i = i + 1)
55      {
56          cin>>e;
57          edificios.push_back(e);
```

```
58      }
59
60      auto l = Dijkstra(lista_adyacencia, bomberos);
61      for (int &e: edificios)
62          cout<<e<<" "<<l[e]<<"\n";
63  }
```

Proposición 25. La complejidad del Algoritmo de Dijkstra sobre un digrafo con pesos $G = (V, E, \psi, w)$ es $O((|E| + |V|) \cdot log|V|)$.

Demostración. Hay que aclarar que, para obtener esta complejidad, es importante manejar la estructura de datos correcta, como una cola de prioridad, que nos ayudará a obtener, en cada iteración, el vértice cuya etiqueta sea la menor, esto en tiempo $O(log|V|)$. Ahora bien, se requiere pasar por todos los vértices (cuando ingresan a P), lo que implica $|V|$ iteraciones, llegando a una complejidad de $O(|V| \cdot log|V|)$.

Lo anterior no es todo, ya que cada vez que un vértice actualiza las etiquetas de sus vecinos, lo hace accediendo a la estructura de datos usada para obtener el mínimo $l(v)$, eso implica otro $O(log|V|)$, lo que se llevará a cabo tantas veces como aristas incidentes haya, es decir, $d(v)$, y considerando que este procedimiento se aplicará a todos los vértices, tendremos una complejidad de $O(|E| \cdot log|V|)$.

Finalmente, la complejidad del algoritmo de Dijkstra será:

$$O(|V| \cdot log|V|) + O(|E| \cdot log|V|) = O(log|V| \cdot (|V| + |E|))$$

□

Con todo lo anterior, hemos comprendido cómo funciona el algoritmo de Dijkstra. Los siguientes resultados nos ayudarán a entender por qué funciona.

Lema 2. Sea p una trayectoria en un grafo con pesos G de v_i a v_j y sea q una subtrayectoria que forma parte de p. Si p es una trayectoria de peso mínimo, entonces q también lo es.

Demostración. Llamemos u y w a los vértices de inicio y de final de q, respectivamente. Luego, p es la unión de las subtrayectorias de v_i a u, de q (que va de u a w) y de

w a v_j, así, el peso de p será equivalente a la suma de los pesos de dichas trayectorias.

Supongamos ahora que existe un camino de menor peso de u a w que q, al que nos referiremos como q'. Si sustituimos a q por q' en p, obtendremos una trayectoria de menor peso que p, lo que es una contradicción de la hipótesis, por lo que no puede ocurrir que q' sea menor que q.

□

Definición 20. Denotamos como $\delta(u, v)$ al peso de la trayectoria de peso mínimo del vértice u al vértice v.

Lema 3. Sea $G = (V, E, \psi, w)$ y s un vértice en V. Para toda arista $v_i v_j$ en E se satisface que $\delta(s, v_j) \leq \delta(s, v_i) + w(v_i, v_j)$.

Demostración. Si p es la trayectoria de peso mínimo de s a v_j, no existe una trayectoria menor, o sea que el peso de p es menor o igual que el peso de alguna trayectoria que va de s a v_i más el peso de la arista (v_i, v_j).

□

Lema 4. Para un digrafo con pesos G sobre el que se ejecuta el algoritmo de Dijkstra se cumple que, para un vértice fuente s y cualquier vértice v, $l(v) \geq \delta(s, v)$ en cualquier momento.

Demostración. Procederemos a demostrar el lema por inducción.

En la primera iteración, es claro que la propiedad se cumple, pues la etiqueta de s, así como el peso de s a s, son 0.

Luego, supongamos que para cierto vértice x se cumple que $l(x) \geq \delta(s, x)$. Cuando x es agregado a P, las etiquetas de los vecinos de x, tales que $l(v) > l(x) + w(x, v)$, $v \in N(x)$, son modificadas, de otro modo, no sufrirán ningún cambio.

Fijémonos pues en los casos específicos en los que existe ese cambio y $l(v)$ adopta el valor de:

$$l(v) = l(x) + w(x, v) \tag{3.1}$$

De la hipótesis de inducción, sabemos que $l(x) \geq \delta(s, x)$, y sustituyendo en 3.1,

$$\begin{aligned} l(v) &= l(x) + w(x, v) \\ &\geq \delta(s, x) + w(x, v) \end{aligned}$$

Finalmente, por el lema 3, $\delta(s, x) + w(x, v) \geq \delta(s, v)$, y por transitividad, $l(v) \geq \delta(s, v)$.

□

Los lemas anteriores nos llevan a poder probar el siguiente teorema, que demuestra que el algoritmo de Dijkstra efectivamente obtiene la ruta mínima desde un vértice fuente hasta cualquier otro para el que existe un camino.

Teorema 17. [Correctitud del Algoritmo de Dijkstra]. Si el algoritmo de Dijkstra se ejecuta sobre un grafo dirigido con pesos no negativos desde un vértice fuente s, entonces, para cada vértice v que es añadido a P se cumple que:

$$l(v) = \delta(s, v)$$

Demostración. Demostraremos el teorema por inducción.

En la primera iteración, cuando $v = s$, sucede que $l(v) = \delta(s, v) = 0$ y el teorema se cumple.

Supongamos ahora que el teorema se cumple para los primeros $k - 1$ vértices que ingresan a P, lo que implica que para cada uno de esos vértices, $l(u) = \delta(s, u)$. Supongamos también que el k-ésimo vértice será v.

Procedamos ahora por contradicción suponiendo que $l(v) \neq \delta(s, v)$, más precisamente, que $l(v) > \delta(s, v)$, pues, por el lema 4, sabemos que no puede pasar que $l(v) < \delta(s, v)$.

Luego, sea q el camino de peso mínimo de s a v. Ya que $s \in P$ y $v \in V - P$, existe una arista xy sobre q tal que $x \in P$ y $y \in V - P$. Ahora bien, cuando x ingresó a P actualizó las etiquetas de sus vecinos, en particular, la de y, ocasionando que:

$$l(y) \leq l(x) + w(x, y) \tag{3.2}$$

(Si $l(x) + w(x, y) < l(y)$, entonces $l(y)$ adoptará el valor de $l(x) + w(x, y)$, de otro modo $l(y)$ permanece igual porque $l(x) + w(x, y) \geq l(y)$).

De la hipótesis de inducción, sabemos que, para los primeros $k - 1$ vértices, $l(u) = \delta(s, u)$, por lo que eso es cierto para x. Además, por el lema 2, el segmento de q que va de s a y es también una trayectoria óptima que incluye a x, por lo que $\delta(s, y) = l(x) + w(x, y)$. Sustituyendo en 3.2: $l(y) \leq \delta(s, y)$. Por otro lado, por el lema 4, $l(y) \geq \delta(s, y)$. Y la única manera de que se cumplan ambas expresiones es que $l(y) = \delta(s, y)$. Esto nos dice que $y \neq v$.

Luego, como $y \in q$ (que termina en v), $\delta(s, y) < \delta(s, v)$, por lo que $l(y) < l(v)$, lo que implica que v no puede ser el siguiente vértice que entre a P, pues deberá ser y, mismo que cumple con el teorema.

□

Ejemplo 116. Cierta persona pretende realizar un viaje de una ciudad A a una ciudad B, y lo hará en auto pasando por varias ciudades más con el fin de conocer mejor su país; sin embargo, por ser viajero frecuente, una aerolínea le ha otorgado un viaje gratis que él deberá elegir de entre una lista de opciones que la misma aerolínea le proporcionará.

Si el viajero te brinda los costos que implicaría recorrer el camino de cada par de ciudades por los que podría pasar, ayúdale a encontrar el precio que implica elegir la ruta más barata. (Problema modificado, tomado de [21]).

> **Solución.** Abstrayendo el problema a una gráfica en la que los vértices representan las ciudades y las aristas los caminos que se pueden recorrer, tal vez tu primera idea haya sido asignarle peso 0 a las aristas que representan los vuelos, sin embargo esta idea tiene un problema: no se pueden agregar todas las aristas con peso 0, pues únicamente es posible elegir una, y agregar todas simultáneamente ocasionaría que se escojan todas a la vez.
>
> Otra posibilidad sería correr un Dijkstra para cada opción de vuelo gratis, de esta forma únicamente habría una arista con peso 0 a la vez; esta opción resolvería el problema de manera correcta, pero no eficiente, pues la complejidad del algoritmo sería $O((|E| + |V|) \cdot log|V| \cdot F)$, en donde F

es el número de vuelos gratis posibles. Con un gran número de vuelos, y dependiendo de las cardinalidades de V y E, corremos el riesgo de que la solución no pase en tiempo.

Buscando otra alternativa para resolver el ejercicio, pensemos en cada vuelo como una arista de peso 0 que va de cierto vértice v_i a cierto vértice v_j. Así, el costo de la ruta más barata para ir de A a B usando un vuelo será:

$$\delta(A, v_i) + w(v_i, v_j) + \delta(v_j, B) = \delta(A, v_i) + \delta(v_j, B)$$

Es fácil ver que, para obtener $\delta(A, v_i)$, únicamente es necesario correr el algoritmo de Dijkstra a partir de A y ubicar el peso correspondiente a su ruta hasta v_i. Por otro lado, no parece buena idea correr el algoritmo de Dijkstra desde v_j a B pues v_j es variable, lo que sí podemos hacer es ejecutar el algoritmo a partir de B para obtener la ruta óptima desde ese vértice hasta cualquier otro.

Finalmente, sólo se requiere obtener la suma de las rutas óptimas $\delta(A, v_i)$ y $\delta(B, v_j)$ con la información arrojada por los algoritmos previamente ejecutados para cada vuelo posible, con lo que la complejidad del procedimiento será $O((|E|+|V|) \cdot log|V|+F)$. Ahora bien, existe la posibilidad de que la ruta óptima de A a B no requiera el uso de ningún vuelo, opción que se considerará al asegurarse de que $\delta(A, B) > \delta(A, v_i)+\delta(B, v_j)$, para todas las posibilidades de vuelo, en cuyo caso la respuesta será simplemente $\delta(A, B)$.

```
1  int solucion(vector<vector<pair<int, int>>> lista_adyacencia, int A,
   ↪  int B, vector<pair<int, int>> Vuelos)
2  {
3      //Dijkstra con origen A.
4      auto lA = Dijkstra_peso(lista_adyacencia, A);
5
6      //Dijkstra con origen B.
7      auto lB = Dijkstra_peso(lista_adyacencia, B);
8
9      int r_min = lA[B];
10     for (auto par: Vuelos)
11         r_min = min(ruta, lA[par.first] + lB[par.second]);
```

```
12
13      return r_min;
14  }
```

Algoritmo de Floyd-Warshall

Aunque el algoritmo de Dijkstra es útil, si lo requerido es obtener los caminos de menor peso a partir de varios vértices, este método no resultará tan eficiente, pues habrá que ejecutarlo una vez por cada vértice. Para ello, la alternativa ideal es lo que se conoce como Algoritmo de Floyd-Warshall.

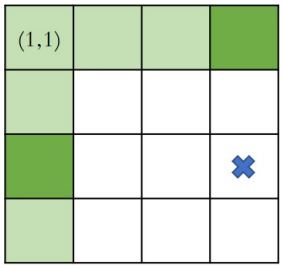
Éste basa su lógica en la matriz de adyacencia $M = [m_{ij}]$, correspondiente al grafo sobre el que se corre el algoritmo y una matriz adicional $P = [p_{ij}]$, cuya función es registrar las rutas óptimas generadas.

Hay que mencionar que la matriz de adyacencia para un grafo con pesos tiene una diferencia importante con las antes vistas, pues ésta no será binaria, sino una matriz $M = [m_{ij}]$ en la que el elemento de la posición (i, j) es el peso asociado a la arista que va del vértice i al vértice j.

Retomando el tema del funcionamiento del algoritmo, establecemos la matriz de adyacencia del grafo con pesos, con la particularidad de que, a los vértices que no son adyacentes, se les asigna un peso mucho más alto que al resto de las aristas (en la teoría, de ∞); la matriz P se llena de tal forma que $[p_{ij}] = i$. Hecho esto, para cada iteración tomaremos un pivote en $M = [m_{ij}]$ tal que $i = j$, razón por la que habrá que hacer $|V|$ iteraciones; pensemos, sin pérdida de generalidad, en $i = j = 1$. Fijaremos nuestra atención en la fila y la columna que pasan por dicho pivote, y para la suma de cada par de elementos $m_{i1} + m_{1j}$, (uno de la fila y uno de la columna), preguntamos si es menor que m_{ij} (el elemento en que se intersectan la fila i y la columna j). Si efectivamente es menor, sustituiremos el valor de m_{ij} por $m_{i1} + m_{1j}$, de lo contrario no se realizará ningún cambio y continuaremos con el procedimiento. Ahora bien, únicamente si en M se realizaron modificaciones se deberá alterar a P, poniendo en p_{ij} el valor de p_{1j} (que es, en realidad, el nombre del

vértice pivote), en este caso 1, lo que indica que, hasta ese momento, la ruta óptima del vértice i al vértice j pasa por el vértice 1. Este mismo proceso se repetirá para cada pivote que se pueda tomar, es decir, para cada elemento en la diagonal de M.

La siguiente matriz ilustra los elementos involucrados en una iteración del algoritmo de Floyd-Warshall.



El pivote se encuentra en $(1, 1)$, y haremos todos los posibles pares entre los elementos de la columna 1 y la fila 1. Por ejemplo, para el par $(3, 1)$ y $(1, 4)$, habrá que sumarlo y compararlo con $(3, 4)$, marcado con una cruz. Éste valor será sustituido sólo en el caso de que la suma de $(3, 1) + (1, 4)$ sea menor que $(3, 4)$.

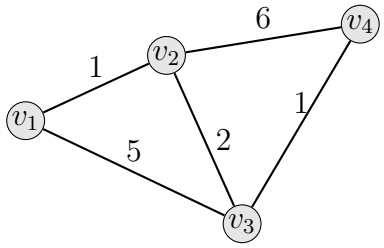
A continuación se presenta el Algoritmo de Floyd-Warshall.

Algoritmo Floyd-Warshall

```
1: Matriz Adyacencia M = [m_ij]
2: Matriz Predecesor P = [p_ij]
3: p_ij = i ∀i, j ∈ |V|
4: para todo k ∈ |V| hacer
5:     para todo i ∈ |V| hacer
6:         para todo j ∈ |V| hacer
7:             si  m_ij > m_ik + m_kj  entonces
8:                 m_ij ← m_ik + m_kj
9:                 p_ij ← p_kj
10:            fin si
11:        fin para
12:    fin para
13: fin para
```

Para reconstruir la ruta tenemos a P. Si queremos ir de cierto vértice i a cierto vértice j nos fijamos en la posición (i, j). Digamos que contiene un número r. Esto nos dice que la ruta óptima para ir de i a j pasa por r. Luego revisamos las posiciones (i, r) y (r, i). Si el dato que en ellos está es igual a alguno de los que ya se han analizado, habremos encontrado todos los vértices que conforman la ruta óptima, de lo contrario, seguimos revisando posiciones de P. Lo veremos con mayor claridad en el siguiente ejemplo.

Ejemplo 117. Aplica el algoritmo de Floyd-Warshall al siguiente grafo.



Solución. Nota primero que estamos tratando con un grafo no dirigido, lo que implica que su matriz de adyacencia será simétrica. Así, construyendo a M y a P:

$$M = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 5 & \infty \\ 1 & 0 & 2 & 6 \\ 5 & 2 & 0 & 1 \\ \infty & 6 & 1 & 0 \end{bmatrix} \end{array} \qquad P = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix} \end{array}$$

Todos los vértices son adyacentes al resto a excepción de v_1 y v_4, y es debido a la ausencia de esa arista que $m_{4,1} = m_{1,4} = \infty$.

En cuanto a P, cada fila es llenada con el número de cada vértice. Si los vértices fueran nombrados de otra forma, digamos, con letras, con letras deberá ser llenada P.

Comenzando con las iteraciones, cuando $k = 1$:

$$M^{[1]} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 5 & \infty \\ 1 & 0 & 2 & 6 \\ 5 & 2 & 0 & 1 \\ \infty & 6 & 1 & 0 \end{bmatrix} \end{array} \qquad P^{[1]} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix} \end{array}$$

En la primera iteración no hubo ninguna modificación, lo que se puede interpretar como que, para cada par de vértices adyacentes v_i, v_j, la mejor opción es simplemente hacer uso de la arista v_iv_j y no de un camino alterno (de dos aristas).

Para la segunda iteración, $k = 2$:

$$M^{[2]} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & 7 \\ 1 & 0 & 2 & 6 \\ 3 & 2 & 0 & 1 \\ 7 & 6 & 1 & 0 \end{bmatrix} \end{array} \qquad P^{[2]} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 3 & 3 & 3 \\ 2 & 4 & 4 & 4 \end{bmatrix} \end{array}$$

Aquí hubo dos cambios:

- En $M^{[1]}$, $(1,3) > (1,2)+(2,3)$ porque $5 > 1+2$, por lo que el elemento en $(1,3)$ toma el valor de 3. Lo mismo aplica para el elemento $(3,1)$, que, gracias a que M es simétrica, es igual a 3 también.
- El segundo cambio ocurre en la posición $(1,4)$ (y $(4,1)$), ya que $7 < \infty$.

Las dos modificaciones anteriores conllevan modificaciones también en P: $(1,3) = (1,4) = (3,1) = (4,1) = 2$. Esto nos dice que, hasta ahora, el camino de menor peso de v_1 a v_3 pasa por v_2 y tiene un peso de 3, y el de v_1 a v_4 de 7.

Cuando $k = 3$:

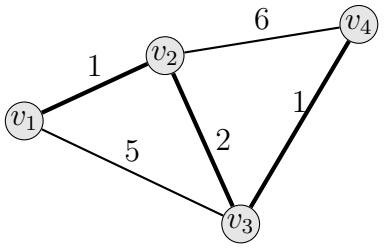
$$M^{[3]} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & 4 \\ 1 & 0 & 2 & 3 \\ 3 & 2 & 0 & 1 \\ 4 & 3 & 1 & 0 \end{bmatrix} \end{array} \qquad P^{[3]} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 2 & 3 \\ 2 & 2 & 2 & 3 \\ 2 & 3 & 3 & 3 \\ 3 & 3 & 4 & 4 \end{bmatrix} \end{array}$$

Las modificaciones de esta iteración se observan en las posiciones $(1,4)$, $(4,1)$, $(2,4)$ y $(4,2)$.

Ya que el grafo tiene 4 vértices, la siguiente iteración es la última, con $k = 4$. En ella no ocurre ninguna modificación.

$$M^{[4]} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & 4 \\ 1 & 0 & 2 & 3 \\ 3 & 2 & 0 & 1 \\ 4 & 3 & 1 & 0 \end{bmatrix} \end{array} \qquad P^{[4]} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 2 & 3 \\ 2 & 2 & 2 & 3 \\ 2 & 3 & 3 & 3 \\ 3 & 3 & 4 & 4 \end{bmatrix} \end{array}$$

Si aún no dominas cómo interpretar los resultados de las matrices, fija tu atención en, por ejemplo, la ruta de v_1 a v_4 (marcada en la gráfica de abajo). En la posición $(1,4)$ de M podemos ver que el peso de dicha ruta es de 4. Para reconstruir la trayectoria, veamos a P: El número en $(1,4)$ es 3, lo que significa que habrá que pasar por el vértice 3; luego, vemos la posición $(1,3)$, que alberga un 2, y después $(1,2) = 1$. Como 1 ya es parte de la ruta óptima, dejamos de buscar ahí, y obtenemos con ello la trayectoria $v_1 - v_2 - v_3 - v_4$. De la misma manera se reconstruye cada ruta.



Proposición 26. Para un grafo $G = (V, E, \psi, w)$ sobre la que se ejecuta el algoritmo de Floyd-Warshall, se cumple que la complejidad del algoritmo es $O(|V|^3)$.

Demostración. Hay tres ciclos de repetición anidados, cada uno de los cuales hace un recorrido sobre todos los vertices, por lo cual se hacen $|V| \cdot |V| \cdot |V| = |V|^3$ comparaciones.

□

Teorema 18. [Correctitud del Algoritmo de Floyd-Warshall]. Sea $G = (V, E, \psi, w)$ un grafo con pesos no negativos sobre el que se corre el algoritmo de Floyd-Warshall, y sean i y j dos vértices en V. En la iteración k del algoritmo se cumple que $m_{ij}^{[k]}$ es el camino de menor peso del vértice i al vértice j utilizando, a lo más, los primeros k vértices, en donde $M^{[k]}$ (que contiene a $m_{ij}^{[k]}$) es M de la k-ésima iteración.

Demostración. Demostrando la correctitud de nuestro algoritmo por inducción fuerte sobre k, sabemos que cuando $k = 0$, se satisface la hipótesis, dado que el peso mínimo de i a j sin vértices intermedios es $w(i, j)$. Probaremos entonces que $m_{ij}^{[k]}$ cumple el teorema si se cumple para la iteración $k - 1$ y todas las anteriores.

Sabemos que $m_{ij}^{[k]} = min\{m_{ij}^{[k-1]}, m_{ik}^{[k-1]} + m_{kj}^{[k-1]}\}$; esto es equivalente a analizar si añadir el vértice k en la ruta concibe una ruta de menor peso que la obtenida en las primeras $k - 1$ iteraciones. Cuando ningún vértice es añadido, sabemos que la ruta encontrada en la iteración anterior es, hasta el momento, la óptima; por otro lado, si un vértice es añadido, ya hemos obtenido la ruta de menor peso de i a k y de k a j, y por el lema 2, la unión de ellas es la óptima.

□

Por la estructura del Algoritmo de Floyd-Warshall, podemos asegurar también que en la k-ésima iteración, los vértices intermediarios de la ruta óptima de i a j (hasta ese momento) pertenecen al conjunto de los primeros k vértices analizados.

Ejemplo 118. Aplica el algoritmo de Floyd-Warshall a un grafo dirigido dado.

> **Solución.** La única diferencia con el ejemplo visto antes será que M no será simétrica. Así, si existe la arista v_iv_j pero no la v_jv_i, se cumplirá que $m_{ij} = w(i, j)$ y $m_{ji} = \infty$, lo que se reflejará en el almacenamiento del

grafo, mas no en el algoritmo de Floyd-Warshall, es decir, el algoritmo es exactamente el mismo para ambos casos.

```
void FloydWarshall(vector<vector<int>> &M, vector<vector<int>> &P)
{
    int k, i, j, n = M.size();
    for (k = 1; k <= n; k = k + 1)
        for (i = 1; i <= n; i = i + 1)
            for (j = 1; j <= n; j = j + 1)
                if (M[i][k] + M[k][j] < M[i][j])
                {
                    M[i][j] = M[i][k] + M[k][j];
                    P[i][j] = P[k][j];
                }
}
```

Ejemplo 119. Aplica el algoritmo de Floyd-Warshall a cierto grafo dado y, de entre todas las rutas de menor peso encontradas, devuelve la de mayor peso. Si hay más de una, devuelve aquella cuyos i, j sean los menores.

Solución. Aplicaremos el algoritmo de Floyd-Warshall a la gráfica dada, ubicaremos el máximo valor de M después de la última iteración y reconstruiremos la ruta que le corresponde.

```
vector<int> r;

void reconstruir_ruta(int i, int j)
{
    if (i == P[i][j])
    {
        r.push_back(i);
        r.push_back(j);
        return;
    }
    reconstruir_ruta(i, P[i][j]);
    reconstruir_ruta(P[i][j], j);
}

vector<int> solucion(vector<vector<int>> &M, vector<vector<int>> &P)
{
    FloydWarshall(M, P);
```

```
18      int i, j, n = M.size(), ruta_max = M[1][1], i1 = 1, j1 = 1;
19      vector<int> r, ruta;
20
21      for (i = 1; i <= n; i = i + 1)
22          for (j = 1; j <= n; j = j + 1)
23              if (ruta_max < M[i][j])
24              {
25                  ruta_max = M[i][j];
26                  i1 = i;
27                  j1 = j;
28              }
29
30      reconstruir_ruta(i1, j1);
31
32      //Para eliminar vértices repetidos.
33      ruta.push_back(r[0]);
34      for (i = 1; i < r.size(); i = i + 1)
35      {
36          if (r[i] == r[i - 1])
37              continue;
38
39          ruta.push_back(r[i]);
40      }
41
42      return ruta;
43  }
```

Ejemplo 120. Un repartidor debe ir todos los días desde un punto A hasta un punto B, ambos fijos. Considerando eso y que todos los días debe entregar exactamente 3 objetos en 3 puntos distintos p_1, p_2 y p_3 (ternas que pueden ser distintas cada día y no necesariamente deben ser adyacentes), su empresa busca la manera de que, cada día, recorra una ruta que permita minimizar los gastos. Ayúdales a encontrar la cantidad de dinero que representaría disponer de la trayectoria óptima.

> **Solución.** Es fácil deducir que buscar la ruta óptima de A a B pasando por p_1, p_2 y p_3 (no necesariamente en ese orden) implica encontrar la ruta de menor peso que permita recorrer a p_1, p_2 y p_3, llegar de A a alguno de esos tres puntos e ir de determinado p_i hasta B. Podríamos pensar en una solución similar a la del ejemplo 116, sin embargo, tenemos el extra de que cada consulta (es decir, cada terna $< p_1, p_2, p_3 >$) implica obtener una ruta óptima entre los p_i's, lo que requiere un Dijkstra para cada po-

sibilidad. Así pues, es mejor obtener desde el principio las rutas óptimas entre cada par de vértices en la gráfica, o sea, introducir el algoritmo de Floyd-Warshall.

En lo que respecta a la ruta óptima entre los p_i's, deberemos analizar cada posibilidad para pasar por todos ellos. Esto es:

$$p_1, p_2, p_3$$
$$p_1, p_3, p_2$$
$$p_2, p_1, p_3$$
$$p_2, p_3, p_1$$
$$p_3, p_1, p_2$$
$$p_3, p_2, p_1$$

Luego, hay que revisar $\delta(A, p_1)$, $\delta(A, p_2)$ y $\delta(A, p_3)$, y $\delta(p_1, B)$, $\delta(p_2, B)$ y $\delta(p_3, B)$; finalmente, habrá que encontrar la unión de rutas óptimas de menor tamaño y ésa será la trayectoria que debe recorrer el repartidor.

```
1  int solucion(vector<vector<int>> &M)
2  {
3      int A, B, p1, p2, p3, r_min;
4      cin>>A>>B;
5      cin>>p1>>p2>>p3;
6      FloydWarshall(M, P);
7
8      //Buscamos el mínimo de tomar cada camino diferente.
9      r_min = M[A][p1] + M[p1][p2] + M[p2][p3] + M[p3][B];
10     r_min = min(r_min, M[A][p1] + M[p1][p3] + M[p3][p2] + M[p2][B]);
11     r_min = min(r_min, M[A][p2] + M[p2][p1] + M[p1][p3] + M[p3][B]);
12     r_min = min(r_min, M[A][p2] + M[p2][p3] + M[p3][p1] + M[p1][B]);
13     r_min = min(r_min, M[A][p3] + M[p3][p1] + M[p1][p2] + M[p2][B]);
14     r_min = min(r_min, M[A][p3] + M[p3][p2] + M[p2][p1] + M[p1][B]);
15
16     return r_min;
17 }
```

Algoritmo de Bellman-Ford

Como habrás notado, los algoritmos que hemos visto hasta ahora son útiles para obtener las rutas de menor peso en grafos con pesos no negativos, sin embargo, ¿qué sucede cuando existen aristas con pesos negativo en el grafo en cuestión? Para ello, el método más conveniente es el algoritmo de Bellman-Ford.

Este algoritmo sirve para encontrar la ruta de menor peso desde un vértice *fuente* al resto de los vértices en la gráfica, y tiene cierta similitud con el Algoritmo de Dijkstra, pues también se requiere un grafo dirigido con pesos, y se asignan etiquetas a los vértices: la fuente comienza con 0 y el resto con ∞, además, basa su procedimiento en el mejoramiento de rutas a partir de rutas previamente mejoradas; como diferencias, podemos nombrar que este proceso realiza siempre $|V| - 1$ iteraciones, en cada una de las cuales se intentan mejorar *todas* las etiquetas.

El algoritmo de Bellman-Ford es más lento que el algoritmo de Dijkstra, sin embargo ofrece una ventaja: éste puede ser ejecutado sobre una gráfica con pesos negativos, y encontrará una solución salvo que en la gráfica haya un *ciclo negativo*, es decir, un ciclo cuyos pesos en las aristas sumen algo menor que 0.

El algoritmo se presenta a continuación, y más adelante probaremos su correctitud, con lo que entenderás por qué son ciertas las afirmaciones que hemos hecho en los párrafos previos.

Las primeras tres líneas únicamente son necesarias para establecer las condiciones iniciales del algoritmo; de la cuarta a la décimo tercera línea se buscan las rutas óptimas; finalmente las últimas siete líneas se encargan de verificar que no exista ningún ciclo negativo. Esto último se hace revisando que, después de encontradas las rutas óptimas, no sea mejorable ninguna de ellas.

Algoritmo Bellman-Ford

```
1: l(v) ← ∞ ∀v ∈ V
2: l(s) ← 0
3: pred(s) ← null
4: para i = 1 hasta |V − 1| hacer
5:     para todo u ∈ V hacer
6:         para todo v ∈ N(u) hacer
7:             si l(v) > l(u) + w(u, v) entonces
8:                 l(v) ← l(u) + w(u, v)
9:                 pred(v) ← u
10:            fin si
11:        fin para
12:    fin para
13: fin para
14: para todo u ∈ V hacer
15:    para todo v ∈ N(u) hacer
16:        si l(v) > l(u) + w(u, v) entonces
17:            ¡Ciclo negativo!
18:        fin si
19:    fin para
20: fin para
```

Teorema 19. [Correctitud del Algoritmo de Bellman-Ford]. Sea G un grafo dirigido con pesos sobre el que se ejecuta el algoritmo de Bellman-Ford. En la i-ésima iteración, para todo vértice v para el que $l(v) \neq \infty$, se cumple que $l(v)$ es, a lo más, el peso del camino más corto del vértice fuente s a v con, a lo más, i aristas.

Demostración. Procederemos por inducción.

Antes de comenzar con la búsqueda de las trayectorias óptimas, s es el único vértice con etiqueta distinta a ∞. En este caso, el teorema se satisface porque el costo para ir de s a s es de 0, utilizando 0 aristas.

Pensemos ahora en el camino de menor peso de s a v, al que nombraremos p, y pensemos en el vértice previo a v, al que nos referiremos como u. Sabemos que el

camino de s a u es óptimo (de otra forma, por el lema 2, podríamos encontrar un camino mejor que p contradiciendo nuestra elección). Supongamos entonces que en la iteración $i-1$, cuando ya hemos encontrado el camino óptimo de s a u, $l(u)$ es, a lo más, el peso del camino más corto de la fuente a u con, a lo más, $i-1$ aristas.

En la i-ésima iteración se requiere comparar $l(v)$ contra $l(u)+w(u,v)$; la etiqueta de v se modificará por $l(u)+w(u,v)$ si esta suma es menor, de lo contrario permanecerá de la misma forma. Así, después de la i-ésima iteración, $l(v)$ iguala el peso de p, y ha ocupado, a lo más, $i-1+1=i$ aristas.

□

Proposición 27. Si después de encontradas las rutas óptimas de cierto digrafo con pesos G, los valores de las etiquetas en los vértices no cambian (líneas 14-20 del algoritmo de Bellman-Ford), entonces no existen ciclos negativos en el grafo.

Demostración. Nota que lo que significan los dos ciclos *for* en esas líneas es que realizaremos una validación dentro de ellos para cada arista.

Pensemos, en particular, en un ciclo con vértices $v_0 - v_1 - \cdots - v_k = v_0$. Si el ciclo es no negativo, para cada arista (v_{i-1}, v_i) dentro del ciclo, *no* se cumplirá que $l(v_i) > l(v_{i-1}) + w(v_{i-1}, v_i)$, o lo que es lo mismo, se satisfacerá que $l(v_i) \leq (v_{i-1}) + w(v_{i-1}, v_i)$. Sumando todas las etiquetas en el ciclo:

$$\sum_{i=1}^{k} l(v_i) \leq \sum_{i=1}^{k} \Big[l(v_{i-1}) + w(v_{i-1}, v_i) \Big]$$

$$= \sum_{i=1}^{k} l(v_{i-1}) + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Pero $\sum_{i=1}^{k} l(v_i) = \sum_{i=1}^{k} l(v_{i-1})$ (pues la primera suma va de v_1 a v_k $(= v_0)$, y la segunda va de v_0 a v_{k-1}). Por tanto:

$$\cancel{\sum_{i=1}^{k} l(v_i)} \leq \cancel{\sum_{i=1}^{k} l(v_{i-1})} + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Restando la suma de las etiquetas de ambos lados de la desigualdad:

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Lo que implica que la suma de los pesos de las aristas dentro del ciclo es no negativa.

□

La proposición anterior nos ayuda a ver que si existe un ciclo negativo, aun después de las $|V| - 1$ iteraciones habrá etiquetas que serán mejorables, lo que implica que la salida del algoritmo no es una solución, pues, en realidad, no hay una solución.

Proposición 28. La complejidad de ejecutar el algoritmo de Bellman-Ford sobre un digrafo con pesos $G = (V, E, \phi, w)$ es $O(|V| \cdot |E|)$.

Demostración. El ciclo que comienza en la línea 4 del algoritmo se repite $|V| - 1$ veces, y para cada una de esas ocasiones, se revisa la validación de la línea 7 para cada arista, lo que se resume en $O((|V| - 1) \cdot |E|)$.

Luego viene la búsqueda de ciclos negativos, que se realiza sobre todas las aristas, esto implica $|E|$ operaciones.

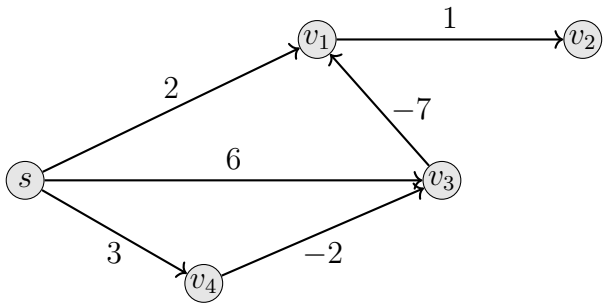
Sumando ambos procesos, tenemos una complejidad de:

$$\begin{aligned} O((|V| - 1) \cdot |E|) + O(|E|) &= O(|E| \cdot (|V| - 1 + 1)) \\ &= O(|E| \cdot |V|) \end{aligned}$$

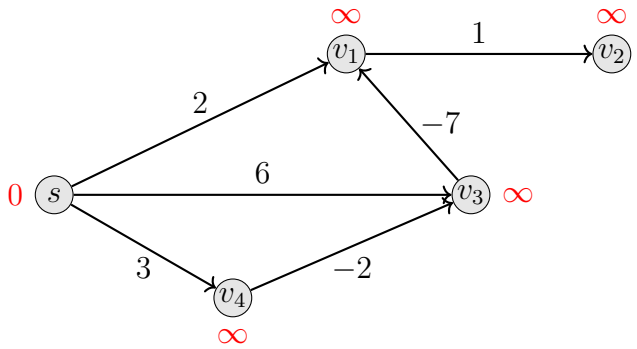
Obteniendo así la complejidad propuesta.

□

Ejemplo 121. Aplica el algoritmo de Bellman-Ford al siguiente grafo.

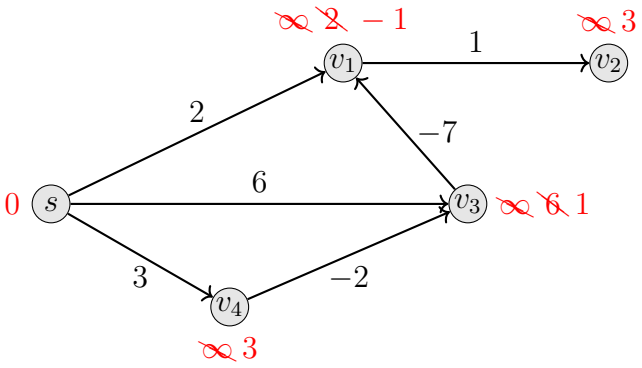


Solución. Recordemos que se haremos $|V|-1=4$ iteraciones, pero antes de ello, hay que asignar las etiquetas a los vértices.



Establecidas las etiquetas previas, podemos comenzar con las iteraciones.

Iteración 1:

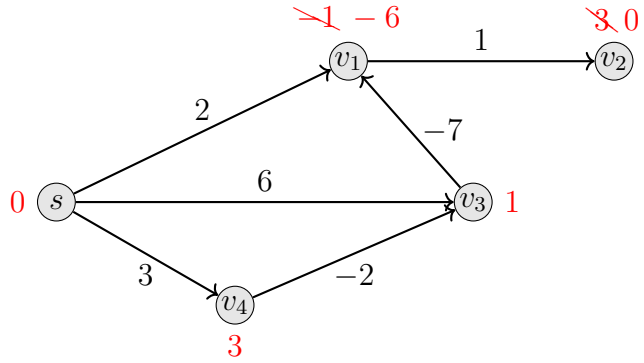


Comenzando por la arista sv_1, la etiqueta de v_1 se verá modificada a 2; este cambio se propagará al vértice v_2 a través de la arista v_1v_3, dándole un valor de $2+1=3$; luego veremos la arista sv_3, que asignará el valor de 6 a v_3; esto a su vez nos regresa al vértice v_1, transformando $l(v_1)$ a -1 puesto que $2>6+(-7)=-1$; finalmente nos centramos en sv_4 para establecer la etiqueta de v_4 en 3, ocasionando un cambio más en v_3 dado que $6>3-2=1$. Considerando lo anterior:

$pred(s)=null$, $pred(v_1)=v_3$, $pred(v_2)=v_1$, $pred(v_3)=v_4$, $pred(v_4)=s$

Iteración 2:

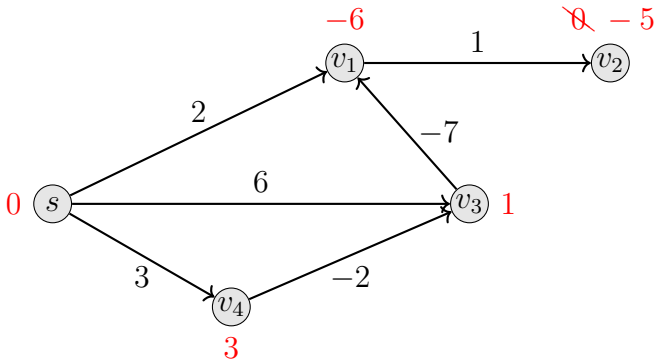
Siguiendo el mismo orden que en la primera iteración:



$pred(s) = null$, $pred(v_1) = v_3$, $pred(v_2) = v_1$, $pred(v_3) = v_4$, $pred(v_4) = s$

Nota que en esta iteración únicamente las etiquetas de dos vértices se vieron modificadas.

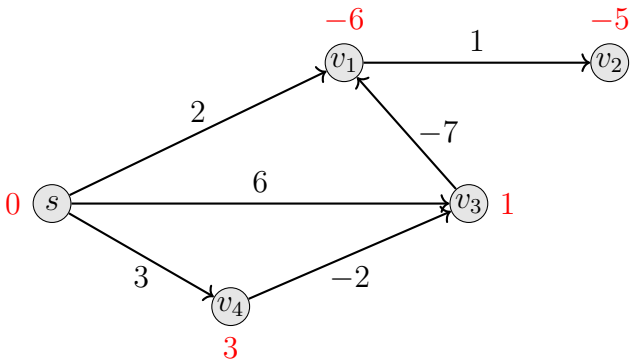
Iteración 3:



$pred(s) = null$, $pred(v_1) = v_3$, $pred(v_2) = v_1$, $pred(v_3) = v_4$, $pred(v_4) = s$

En la tercera iteración sólo se registra el cambio de $l(v_2)$.

Iteración 4:

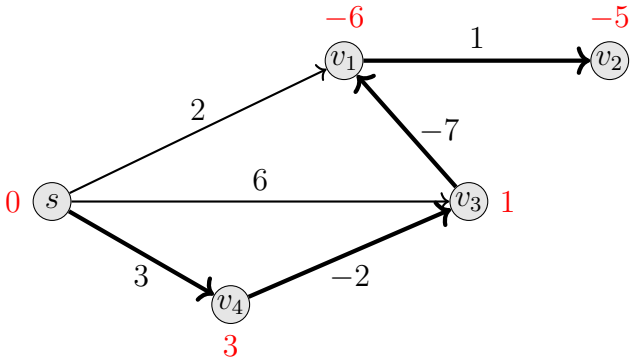


$pred(s) = null$, $pred(v_1) = v_3$, $pred(v_2) = v_1$, $pred(v_3) = v_4$, $pred(v_4) = s$

En ésta, la última pasada, ya no fue necesario ningún cambio.

En este punto, las rutas óptimas pueden ser generadas utilizando los predecesores (de la misma forma que en el algoritmo de Dijkstra), sin embargo, primero es necesario cerciorarse de que no existan ciclos negativos; esto se hará volviendo a revisar que para ninguna arista v_iv_j se cumpla que $l(v_j) > l(v_i) + w(v_i, v_j)$. Para fines prácticos, ya que en la cuarta iteración no hubo cambios, podemos asegurar que no hay ciclos negativos.

Reconstruyendo las rutas óptimas:



Ejemplo 122. Codifica el algoritmo de Bellman-Ford.

Solución.

```cpp
vector<int> l, pred;

void BellmanFord(vector<vector<pair<int, int>>> &lista_adyacencia, int
 ↪  s)
{
    int V = lista_adyacencia.size(), i, u;
    l.resize(V, INF);
    l[s] = 0;
    pred.resize(V);

    for (i = 1; i <= V - 1; i = i + 1)
        for (u = 1; u <= V; u = u + 1)
            for (auto &par: lista_adyacencia[u])
            {
                int v = par.first;
                int w_uv = par.second;
                if (l[v] > l[u] + w_uv)
                {
                    l[v] = l[u] + w_uv;
                    pred[v] = u;
                }
            }
}

bool ciclo_negativo(vector<vector<pair<int, int>>> &lista_adyacencia)
{
    int V = lista_adyacencia.size(), u;
    for (u = 1; u <= V; u = u + 1)
        for (auto &par: lista_adyacencia[u])
        {
            int v = par.first;
            int w_uv = par.second;
            if (l[v] > l[u] + w_uv)
                return true;
        }
    return false;
}
```

Ejemplo 123. Cierto vendedor de libros a domicilio debe ir de un sitio A a un sitio B pasando por varios puntos; algunos de ellos son de compra y otros son de venta, pero en todos ellos debe realizar algún movimiento que le costará cierta cantidad de dinero $-c_i$ o le remunerará c_i. Ayúdale a encontrar las ganancias que le implicará seguir la ruta óptima (la que le permita maximizar las ganancias).

Solución. Nota que cuando utilizamos el algoritmo de Bellman-Ford sobre un grafo obtenemos la ruta de menor peso, pero lo que queremos hacer en este ejercicio es maximizar. Para remediar esta situación, pensemos en que, en lugar de restar los gastos y sumar las ventas para maximizar las ganancias, podemos sumar los gastos y restar las ventas para minimizar las pérdidas. Así pues, antes que nada deberemos multiplicar por -1 cada c_i (o $-c_i$) para luego aplicar el algoritmo de Bellman-Ford. Ahora bien, como modificamos el signo de las entradas, también deberemos modificar el signo de la salida, y esa será la respuesta.

Otro punto que es importante mencionar es que los pesos están dados en los vértices, por lo que toda arista para la que un vértice sea el vértice de llegada recibirá el mismo peso.

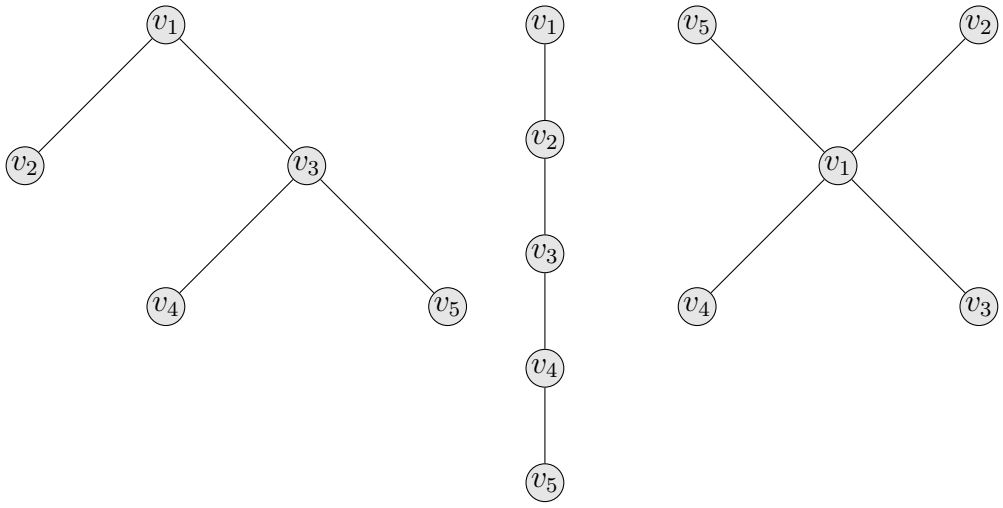
```
int solucion(vector<vector<pair<int, int>>> &lista_adyacencia)
{
    int V, E, i, u, v, A, B;
    cin>>V>>E>>A>>B;
    vector<int> c(V + 1);
    lista_adyacencia.resize(V + 1);

    for (i = 1; i <= V; i = i + 1)
    {
        cin>>c[i];
        c[i] = -c[i]; //Cambio de signo en los costos.
    }

    //Leer topología de la gráfica.
    for (i = 1; i <= E; i = i + 1)
    {
        cin>>u>>v;
        lista_adyacencia[u].push_back({v, c[v]});
    }

    BellmanFord(lista_adyacencia, A);
    return -l[B]; //Ganancias.
}
```

3.5. Árboles

Definición 21. Un *árbol* es un grafo simple conexo que no contiene ciclos. Este tipo de grafos suelen denotarse por la letra T.

Definición 22. Un *bosque* es un grafo acíclico compuesto de uno o varios árboles que no necesariamente es conexo.

Ejemplo 124. Los grafos mostrados a continuación son ejemplos de árboles. Si estos fueran parte del mismo grafo, estaríamos tratando con un bosque.



Definición 23. Sea $T = (V, E, \psi, w)$ un árbol. Llamamos *raíz* a un vértice $v \in V$ cuyo conjunto de predecesores es nulo.

Definición 24. Sea $T = (V, E, \psi, w)$ un árbol. Llamamos *hoja* a un vértice $v \in V$ cuyo conjunto de sucesores es nulo.

Teorema 20. Cualquier par de vértices en un árbol está conectado por un único camino.

Demostración. Sean u y v dos vértices del árbol T. Supongamos que existen dos caminos distintos de u a v, a los que llamaremos p_1 y p_2. Así, sin pérdida de generalidad, existe al menos una arista (x, y) en p_1 que no está en p_2. Claramente el subgrafo $p_1 \cup p_2 - xy$ es conexo, por lo que existe un camino de x a y distinto de la arista xy, lo que implica la existencia de un ciclo en T, contradicción a la definición de árbol. □

Teorema 21. Sea G una gráfica simple y e una hoja de G. G es un árbol si y sólo si $G - v$ también lo es.

Demostración. Establezcamos primero que G es un árbol para probar que $G - v$ lo es: Sean x y y dos vértices distintos en $G - v$. Ya que G es un árbol, es conexo, y por tanto, debe existir un camino p de x a y. Como x y y son las únicas hojas en p, v no es parte de dicho camino, lo que nos lleva a asegurar que $p \subseteq G - v$. Debido a que podemos asegurar que se puede encontrar un camino para cualquier par de vértices en G y seguir el mismo razonamiento de p, deducimos que para cada par de vértices en $G - v$ habrá un camino también, es decir, que $G - v$ es conexa. Además, G es acíclica (por ser un árbol), por lo que $G - v$ también lo es. Con esto probamos que $G - v$ es un árbol.

Vayamos a la segunda parte de la demostración y probemos que G es un árbol cuando $G - v$ lo es: Ya que v es una hoja, al añadirse al grafo $G - v$ (que es acíclico) no habrá ningún ciclo. Sea v' el vértice adyacente a v en G. Para cada par de vértices en $G - v$ existe un camino, en particular, para cualquier x y v', siendo posible extender esta trayectoria a v, en G, lo que implica que G es también un árbol. □

Teorema 22. Para todo árbol $T = (V, E, \psi)$ se cumple que $|E| = |V| - 1$.

Demostración. Procediendo por inducción fuerte sobre V, cuando $|V| = 1$, $|E| = 0$.

Luego, suponemos que el teorema es cierto para todo árbol T no trivial con menos de n vértices y nos enfocaremos en probar que el teorema se satisface para un árbol T_n de n vértices.

Si removemos alguna arista uv de T_n pueden ocurrir dos cosas: que sea una hoja o no. Si es una hoja, por el teorema anterior, podemos garantizar que el grafo resul-

tante sea un árbol también; veamos entonces el otro caso.

Si uv no es una hoja, retirarla ocasionará que se obtengan dos subgrafos no conexos entre sí G_1 y G_2, mismos que son árboles de menos de n vértices, y por la hipótesis de inducción se cumplirá que:

$$|E_{G_1}| = |V_{G_1}| - 1$$
$$|E_{G_2}| = |V_{G_2}| - 1$$

Considerando lo anterior, podemos escribir el número de aristas de T como:

$$E_T = |E_{G_1}| + |E_{G_2}| + 1$$

(El 1 corresponde a la arista uv). Luego,

$$\begin{aligned} E_T &= |E_{G_1}| + |E_{G_2}| + 1 \\ &= (|V_{G_1}| - 1) + (|V_{G_2}| - 1) + 1 \\ &= |V_{G_1}| + |V_{G_2}| - 1 \end{aligned}$$

Y $|V_{G_1}| + |V_{G_2}|$ es la cardinalidad del conjunto de vértices de T.

□

3.5.1. Árbol de Expansión Mínima

Definición 25. Sea un grafo $G = (V, E, \psi, w)$. Se conoce como *árbol de expansión* al árbol $T = (V, E_T, \psi_T)$. Dicho de otra forma, el árbol de expansión de G es el árbol que surge de tomar todos los vértices de G y las aristas necesarias para formar un subgrafo conexo acíclico. Claramente, puede existir más de un árbol de expansión para el mismo grafo.

Definición 26. Sea un grafo $G = (V, E, \psi, w)$. Un *árbol de expansión mínima* es el árbol de expansión de menor peso del grafo G. Éste tampoco es necesariamente único.

Teorema 23. Todo grafo conexo contiene un árbol de expansión.

Demostración. Si el grafo en cuestión no tiene ningún ciclo, éste es su propio árbol de expansión. En caso contrario, para cada ciclo presente en el grafo, es posible retirar una arista, desapareciendo el ciclo y manteniendo la gráfica conectada.

□

Encontrar el árbol de expansión mínima de un grafo tiene toda una gama de aplicaciones tanto en programación competitiva como en la resolución de problemas reales, pues proporciona una manera de cubrir todos los puntos de interés utilizando la configuración menos costosa.

Es importante señalar que un grafo puede tener más de un árbol de expansión mínima, sin embargo, en la mayoría de los casos, esto nos será indiferente, pues no importa cuál de ellos elijamos, tendrán el mismo peso y pasarán por los mismos vértices.

Para localizar el árbol de expansión mínima de un grafo se puede utilizar cualquiera de los dos algoritmos que se describen a continuación.

Algoritmo de Kruskal

El algoritmo de Kruskal sigue un procedimiento basado en el ordenamiento ascendente de sus aristas para luego comprobar si la adición de la siguiente arista en la sucesión forma un ciclo. Si no es el caso, se añade la arista al árbol de expansión; de lo contrario, se ignora la arista y se continúa con la revisión de la siguiente.

Algoritmo Kruskal

1: $E_T \leftarrow \emptyset$
2: Ordenar los aristas E por peso no decreciente: $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$
3: **para todo** arista $e \in E$ **hacer**
4: **si** $T(V, E_T \cup e)$ no forma ciclos **entonces**
5: $E_T \leftarrow E_T \cup e$
6: **fin si**
7: **fin para**

Teorema 24. [Correctitud del Algoritmo de Kruskal]. La salida del algoritmo de Kruskal aplicado a un grafo conexo dado G es un árbol de expansión mínima.

Demostración. Sea Y el grafo de salida del algoritmo de Kruskal. Es necesario demostrar dos cosas: que Y es un árbol de expansión, y que es de peso mínimo.

Para el primer punto, debemos probar que Y es conexo, acíclico y pasa por todos los vértices de G. De entrada, por la naturaleza del algoritmo, no es posible que se creen ciclos. Luego, supongamos que Y es no conexo, y dos componentes C_1, C_2 forman parte de él; así, en G existe un subconjunto de aristas xy tales que x está en C_1 y y en C_2, y cuando el algoritmo de Kruskal analiza la menor de ellas, ésta y únicamente ésta debe ser agregada a Y, pues, al ser la única arista que conecta a C_1 y C_2, no se formará ningún ciclo, haciendo a Y un grafo conexo. Ahora revisemos que Y pase por todos los vértices de G. Supongamos que existe un vértice $v \in V_G$ que no está en Y, lo que implicaría que ninguna de las aristas de la forma xv fue agregada a Y, pero tuvo que ocurrir que la menor de ellas fuera añadida.

Con lo anterior, probamos que Y es un árbol de expansión de G, pero aún necesario probar que Y es, en efecto, mínimo. Sea T un árbol de expansión mínima de G. Si $T = Y$ habremos terminado. Consideremos pues el caso en que esa aseveración es falsa, esto es, cuando existe al menos una arista e_1 en T que no esté en Y; entonces el grafo $Y_1 = Y \cup e_1$ tendrá un ciclo C. Por otro lado, existe una arista $f_1 \in C$ que no pertenece a T. $w(e_1)$ es mayor o igual que cada arista en C (de lo contrario, el algoritmo habría tomado a e_1), en particular,

$$w(e_1) \geq w(f_1) \tag{3.3}$$

La gráfica Y_1, obtenida de la unión de Y y e_1, después de suprimir a f_1, será también un árbol de expansión de peso:

$$w(Y_1) = w(Y) + w(e_1) - w(f_1)$$

De 3.3, $w(e_1) - w(f_1) \geq 0$, y por tanto:

$$\begin{aligned} & w(Y) \leq w(Y) + w(e_1) - w(f_1) \\ \Longrightarrow\ & w(Y) \leq w(Y_1) \end{aligned}$$

Nota que, lo que hicimos, fue agregar al grafo de salida del algoritmo una arista de un árbol de expansión mínima de G. De la misma forma en que construimos Y_1, podemos obtener un subgrafo Y_2 (comparando T contra Y_1), Y_3, y así sucesivamente hasta que todas las aristas de Y_k sean las mismas que las de T, con lo que obtendríamos:

$$w(Y) \leq w(Y_1) \leq w(Y_2) \leq \cdots \leq w(Y_k) = w(T)$$

Pero T es un árbol de expansión mínima, y ya que no puede haber árboles de expansión con un peso menor, sólo puede ocurrir que:

$$w(Y) = w(Y_1) = w(Y_2) = \cdots = w(Y_k) = w(T)$$

Es decir, que Y tenga el mismo peso que un árbol de expansión mínima de G, y sea, por tanto, un árbol de expansión mínima también.

□

Teorema 25. Si $G = (V, E)$ es un grafo con pesos sobre el que se ejecuta el algoritmo de Kruskal. La complejidad de dicha ejecución será $O(|E| \cdot log|E|) = O(|E| \cdot log|V|)$.

Demostración. Hablemos primero del ordenamiento de las aristas, que se realiza en tiempo $O(|E| \cdot log|E|)$ si se utiliza la función *sort* propia del lenguaje. Luego, para examinar si la adición de alguna arista forma un ciclo, podemos hacer uso de una estructura de datos de *union find disjoint-sets* (cuya explicación podrás encontrar en [18]), que permite realizar las consultas en tiempo constante, con lo que concluimos que el tiempo en que se realiza todo el algoritmo es $O(|E| \cdot log|E|)$.

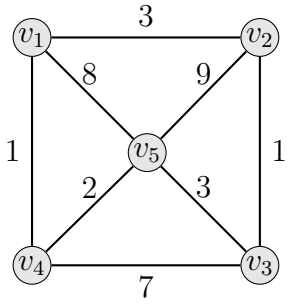
El peor caso se presenta cuando $|E| = |V|^2$ (todos los vértices son adyacentes entre sí). Entonces:

$$O(|E| \cdot log|E|) = O(|E| \cdot log(|V|^2))$$

Pero $log(|V|^2) = 2log|V|$, por lo que $O(|E| \cdot log|E|) = O(|E| \cdot log|V|)$.

□

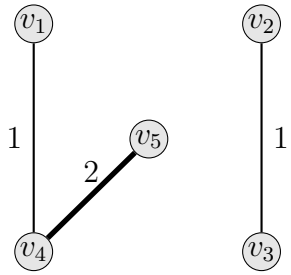
Ejemplo 125. Aplica el algoritmo de Kruskal al siguiente grafo.



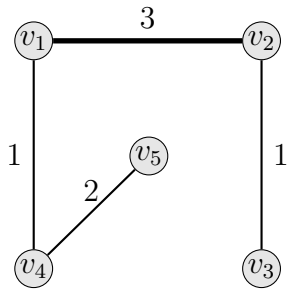
Solución. Ordenando las aristas de manera ascendente por su peso:

$$\begin{aligned}
(v_1, v_4) &\rightarrow 1 \\
(v_2, v_3) &\rightarrow 1 \\
(v_4, v_5) &\rightarrow 2 \\
(v_1, v_2) &\rightarrow 3 \\
(v_3, v_5) &\rightarrow 3 \\
(v_3, v_4) &\rightarrow 7 \\
(v_1, v_5) &\rightarrow 8 \\
(v_2, v_5) &\rightarrow 9
\end{aligned}$$

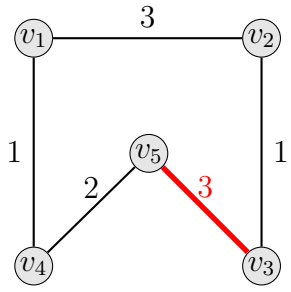
Obviamente tomaremos la primera arista de la lista, y dado que su unión con la segunda no forma ningún ciclo, también tomaremos (v_2, v_3). Seguiremos entonces el análisis a detalle desde la arista (v_4, v_5):



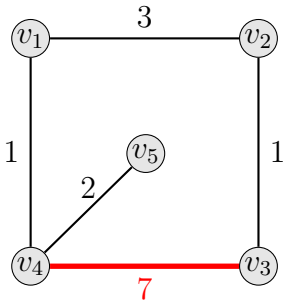
Ya que no hay ciclo, agregamos la arista y continuamos con (v_1, v_2):



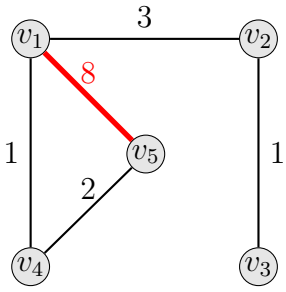
Ya que no hay ciclo, agregamos la arista y continuamos con (v_3, v_5):



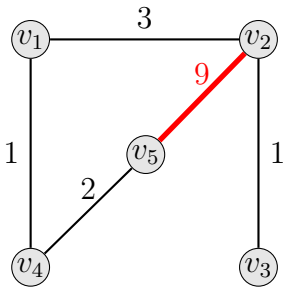
En este punto nos topamos con el primer ciclo, por lo que la arista v_3v_5 no puede ser incluida. Continuamos entonces con la (v_3, v_4):



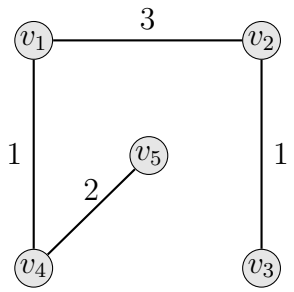
Tenemos otro ciclo. No podemos incluir v_3v_4. Continuamos con (v_1, v_5):



Tenemos otro ciclo, por lo que no podemos incluir a v_1v_5. Finalmente revisamos a la arista (v_2, v_5):



Así concluimos que un árbol de expansión mínima de nuestro grafo tiene un peso de 7 y se muestra a continuación. El grafo obtenido incluye todos los vértices del grafo original y utiliza únicamente las aristas necesarias para conectarlos a todos sin formar un ciclo, esto con el peso mínimo posible.



Nota que, gracias a la estructura del algoritmo, éste es útil no sólo para hallar árboles de expansión mínima, sino también bosques de expansión mínima, pues ninguna condición verifica que las aristas elegidas sean conexas.

Ejemplo 126. Cierta ciudad pretende colocar un nuevo cableado de luz, y en el proceso se han dado cuenta de que el que actualmente poseen utiliza más cables de los necesarios. Por ello, han decidido conservar únicamente los cables que sean indispensables para cubrir todos los postes que se tienen actualmente, de tal forma que el material nuevo sea mínimo. ¿Cuál es la cantidad de cable que se requerirá para cubrir de manera óptima la red del cableado de luz?

Solución. Requerimos encontrar el peso del árbol de expansión mínima del cableado antiguo.

Como mencionamos antes, es necesario el dominio de *union find disjoint sets*, que no será incluido en esta obra, por lo que en el programa únicamente se indicarán las líneas en pseudocódigo que se requerirían para la resolución del problema.

El grafo será guardado previamente en una lista aristas, misma que contendrá también el peso de cada una de ellas.

```
1  int Krusal(vector<pair<int, pair<int, int>>> lista_aristas)
2  {
3      int peso_arbol = 0, w, u, v;
4      sort(lista_aristas.begin(), lista_aristas.end());
```

```
5
6       for (auto &arista: lista_aristas)
7       {
8           w = arista.first; //Peso.
9           u = arista.second.first; //Primer vértice de la arista.
10          v = arista.second.second; //Segundo vértice de la arista.
11
12          //Pseudocódigo.
13          if (not ciclo(u, v))
14          {
15              unir(u, v);
16              peso_arbol = peso_arbol + w;
17          }
18      }
19
20      return peso_arbol;
21  }
```

Algoritmo de Prim-Jarník

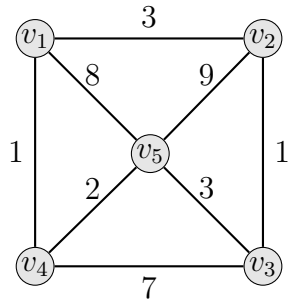
En este algoritmo se elige un vértice fuente arbitrario s como el primer elemento del árbol de expansión T, cuyo conjunto de vértices analizados será P; luego se adhiere la arista de menor peso que incide en s y el otro vértice al que es incidente se agrega a P. Así, continuamos buscando aristas incidentes a un sólo vértice en el conjunto P (si una arista es incidente a dos vértices en P no se debe tomar, pues ésta formaría un ciclo en T).

Este método sigue un razonamiento similar al del ya visto algoritmo de Dijkstra, pues también realiza una asignación de etiquetas inicial en ∞, que se irán actualizando conforme avanza el algoritmo.

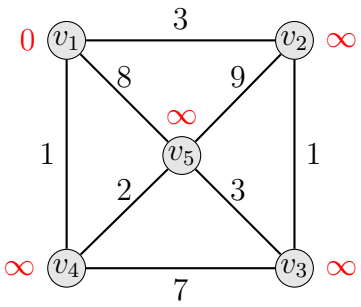
Algoritmo Prim-Jarník

1: $E_T \leftarrow \emptyset$
2: Procesado $P \leftarrow \emptyset$
3: **para cada** $u \in V$ **hacer**
4: $\quad l(u) \leftarrow \infty$
5: **fin para**
6: $l(s) \leftarrow 0$
7: $pred(s) \leftarrow null$
8: **mientras** $P \neq V$ **hacer**
9: $\quad u \leftarrow v \in V - P$ tal que $l(u) = \text{mín}\{l(v) | v \in V - P\}$
10: $\quad P \leftarrow P \cup \{u\}$
11: $\quad E_T \leftarrow E_T \cup \{u, pred(u)\}$
12: $\quad$ **para todo** $v \in N(u) \cap V - P$ tal que $l(v) > w(u, v)$ **hacer**
13: $\quad\quad l(v) \leftarrow w(u, v)$
14: $\quad\quad pred(v) \leftarrow u$
15: $\quad$ **fin para**
16: **fin mientras**

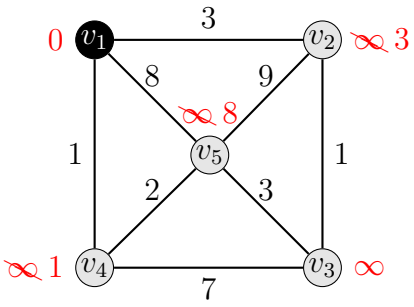
Ejemplo 127. Aplica el algoritmo de Prim-Jarník al siguiente grafo.



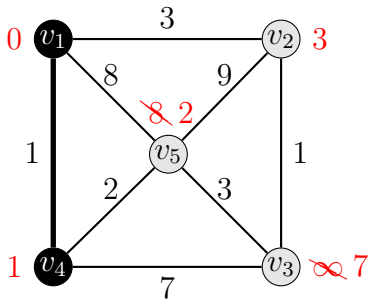
Solución. Estableceremos a v_1 como s, no sin antes de asignar las etiquetas correspondientes.



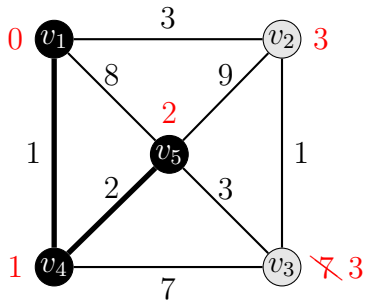
La menor etiqueta es la de $s = v_1$, por lo que será el primer vértice en entrar a P, sustituyendo las etiquetas de todos sus vecinos por el peso de la arista que los conecta.



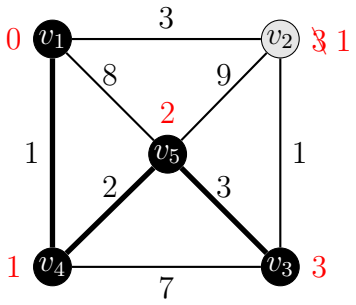
El vértice con la menor etiqueta ahora es v_4, por lo que la primera arista que formará parte del árbol de expansión mínima es (v_1, v_4) (del hecho de que v_1 es el predecesor de v_4). Luego, modificamos las etiquetas de los vértices vecinos a v_4 que no estén ya en P: Enfocándonos en la arista (v_4, v_5), podemos ver que $8 > 2$, por lo que $l(v_5)$ debe ser sustituida por $w(v_4, v_5) = 2$; de igual manera hay que modificar $l(v_3)$ por 7, convirtiendo a v_4 en su predecesor.



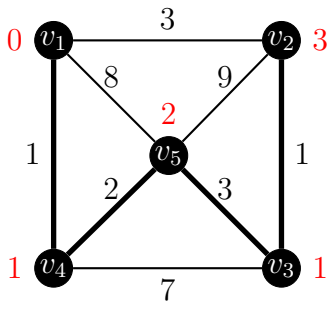
La menor etiqueta entre los vértices no analizados ahora pertenece a v_5, incluyendo así a la arista (v_4, v_5). Este cambio, a su vez, propicia la alteración de la etiqueta de v_3, dándole el valor de 3 y cambiando su predecesor a v_5; v_2 también es un vecino no procesado de v_5, pero este no sufre ningún cambio al ser 3 menor que 9.



Los únicos dos vértices que no se han procesado tienen la misma etiqueta, así que podemos tomar cualquiera de los dos. Nosotros elegiremos primero a v_3, cuyo predecesor es v_5, añadiendo así a la arista (v_5, v_3). Finalmente, el vértice v_2 es el único sin procesar, y v_3 modificará su valor a 1, haciéndose también su predecesor.



Por último, analizamos a v_2 y tomamos la arista $(v_2, pred(v_2)) = (v_2, v_3)$ al grafo de salida. De esta forma, el árbol de expansión del grafo dado es:



Nota que el árbol de expansión mínima que obtuvimos aquí es distinto al del ejemplo 125, pero tienen el mismo peso, lo que prueba que dicho árbol no es necesariamente único.

Teorema 26. [Correctitud del Algoritmo de Prim-Jarník]. El grafo de salida del algoritmo de Prim-Jarník aplicado a un grafo $G = (V, E)$ es un árbol de expansión mínima de G.

Demostración. Realizaremos la prueba por inducción, demostrando que el grafo generado en cada iteración del algoritmo es un subgrafo de algún árbol de expansión mínima T de G.

Es claro que en la primera iteración el teorema es cierto, pues únicamente tenemos un vértice y ninguna arista.

Supongamos que en la penúltima iteración del algoritmo tenemos un grafo Y subgrafo de T.

Hay que demostrar entonces que, al agregar la siguiente arista e, se obtiene también un árbol de expansión de G. Aquí existen dos posibilidades: que $e \in T$ y que $e \notin T$. Si se cumple lo primero, habremos terminado, así que enfoquémonos en el segundo escenario (que se desarrollará de manera muy similar a la demostración de la correctitud del algoritmo de Kruskal): cuando no existe una arista en T que Y incluye. Es claro que $T \cup e$ tiene un ciclo. Por otro lado, existe una arista f que está en dicho ciclo y no está en Y, y cuyo peso es mayor o igual que $w(e)$ (de lo contrario, se habría elegido a f). Luego, creamos el árbol $Y_1 = Y \cup f - e$, y su peso será:

$$w(Y_1) = w(Y) + w(f) - w(e)$$

$$\implies w(Y_1) \geq w(Y)$$

Pues $w(f) \geq w(e)$, y por tanto, $w(f) - w(e) \geq 0$.

Si $Y_1 \neq T$, entonces repetimos el proceso, hasta que todas las aristas en Y_k coincidan con las de T. Con ello, tendremos las siguientes relaciones:

$$w(T) = w(Y_k) \geq \cdots \geq w(Y_2) \geq w(Y_1) \geq w(Y)$$

Y como T tiene el peso mínmo, lo único que puede pasar es que las relaciones anteriores sean igualdades, por lo que $w(T) = w(Y)$.

□

Teorema 27. Si se ejecuta el algoritmo de Prim-Jarník sobre un grafo $G = (V, E)$, éste tendrá una complejidad de $O((|V| + |E|) \cdot log|V|)$.

Demostración. La asignación inicial de las etiquetas se realiza sobre los $|V|$ vértices del grafo, lo que implica $|V|$ operaciones en un arreglo. Por otro lado, tendremos una cola de prioridad para almacenar las etiquetas que sean mejoradas, cuyo primer elemento será s.

Luego entraremos al primer ciclo de repetición, que se realizará $|V|$ veces, en cada una de las cuales deberemos obtener el mínimo (dentro de la cola de prioridad), lo que gracias a la estructura de datos utilizada tendrá una complejidad de $O(log|V|)$ (agregar al vértice y a la arista correspondientes al árbol de expansión mínima son operaciones constantes, así que no resultan relevantes en este caso), a su vez, en cada una de estas operaciones, habrá que revisar si $l(v) > l(u) + w(u, v)$ para todos los vértices adyacentes al vértice actual, esto es, $\forall v \in V$, revisar $d(v)$, que son en realidad todas las aristas de G, es decir, $O(|E|)$, además, si se hace la sustitución de $l(v) = w(u, v)$, tendremos que acceder a la cola de prioridad, lo que se hace en tiempo $O(log|V|)$, con lo que llegaremos a $O(|E| \cdot log|V|)$.

Con lo anterior, la complejidad del algoritmo se resume a:

$$\begin{aligned} &O(|V|) + O(|V| \cdot log|V|) + O(|E| \cdot log|V|) \\ =&O(|V|) + O((|V| + |E|) \cdot log|V|) \end{aligned}$$

Como $|V|$ es menor que el resto de la expresión, podemos prescindir de él, dejando la complejidad en $O((|V| + |E|) \cdot log|V|)$.

□

Ejemplo 128. Codifica el algoritmo de Prim-Jarník.

Solución.

```
int PrimJarnik(vector<vector<pair<int, int>>> lista_adyacencia, int s)
{
    int t, n = lista_adyacencia.size(), procesados = 0, peso_arbol = 0;
    int peso_arista, u, v, w;
    vector<int> pred(n), l(n, INF);
    bitset<10000> procesado;
    priority_queue<pair<int, int>, vector<pair<int, int>>,
                   greater<pair<int, int>>> cola_prioridad;
    l[s] = 0;
    pred[s] = 0;
    cola_prioridad.push({l[s], s});

    n = n - 1;  //Eliminar el vértice 0, no existe.
    while (procesados != n)
    {
        u = cola_prioridad.top().second; //Siguiente vértice.
        peso_arista = cola_prioridad.top().first; //Última etiqueta.
        cola_prioridad.pop();

        if (procesado[u])
            continue;

        procesado[u] = true;
        procesados = procesados + 1;
        peso_arbol = peso_arbol + peso_arista;

        for (auto par: lista_adyacencia[u])
        {
            v = par.first; //Vecino de u.
            if (procesado[v])
                continue;

            w = par.second; //Peso de arista uv.
            if (l[v] > w)
            {
                l[v] = w;
                pred[v] = u;
```

```
38                      cola_prioridad.push({w, v});
39                  }
40              }
41          }
42
43          return peso_arbol;
44      }
```

La complejidad de los dos algoritmos que vimos para encontrar el árbol de expansión mínima es muy similar, por lo que no podemos posicionar a uno sobre otro en complejidad por mucho. No obstante, podemos decir que el algoritmo de Kruskal es útil si el grafo de entrada es no conexo, pues gracias a su funcionamiento, más allá de encontrar árboles de expansión mínima, es capaz de hallar bosques; la desventaja con este método es que, al no almacenar ninguna relación entre los vértices, podremos encontrar el peso, pero no reconstruir el árbol, problema que sí puede resolver el algoritmo de Prim-Jarník.

3.6. Búsquedas en Grafos

Las búsquedas son algunos de los recursos más básicos e importantes de la programación. Éstas permiten recorrer un espacio de posibles soluciones y determinar cuáles de ellas son correctas de acuerdo con lo requerido en cierta situación.

Es importante recalcar que, para aplicar una búsqueda correctamente, es necesario tener claro cuál será la forma de representación que se le dará a los datos trabajados, así como tener un dominio considerable de las técnicas.

El objetivo de las búsquedas es, literalmente, recorrer cada uno de los elementos del grafo y analizarlo, por lo que, en algunas ocasiones, podría no ser la solución óptima. Como ventaja de estas técnicas podemos mencionar que, aplicadas correctamente, pueden garantizar que la salida arrojará los datos correctos, pues califica las características de cada uno para determinar si es uno de los datos buscados o

no, esto claro a costa de un tiempo de ejecución mayor. Por ello, es recomendable analizar primero la cantidad de datos de entrada y el tiempo límite para procesarlos.

En esta sección estudiaremos los métodos de búsqueda en profundidad y búsqueda en amplitud para después abordar algunas de sus aplicaciones más usuales.

3.6.1. Búsqueda en Profundidad

La búsqueda en profundidad, DFS por sus siglas en inglés (deep first search), es un algoritmo recursivo que sirve para visitar los vértices de un componente de un grafo; cada vez que la búsqueda alcanza un vértice no visitado, digamos v, intentará ir más "profundo" visitando algún vecino de v, y lo hará hasta que ya no pueda ir a mayor profundidad. Una vez que haya pasado esto, el algoritmo regresará a visitar otro de los vecinos no visitados de v. A continuación, se muestra el pseudocódigo del proceso:

Algoritmo Búsqueda en Profundidad

```
1: función PREPROCESO(V)
2:     para cada v ∈ V hacer
3:         estado(v) = no visitado
4:     fin para
5: fin función
6: función BÚSQUEDA EN PROFUNDIDAD(v)
7:     estado(v) ← visitado
8:     para cada u ∈ N(v) hacer
9:         si estado(u) = no visitado entonces
10:            Búsqueda en profundidad(u)
11:        fin si
12:    fin para
13: fin función
```

El proceso de la búsqueda requiere pasar por todos los vértices, y para cada uno, revisar sus vecinos, es decir, el número de aristas, por lo que la complejidad del algoritmo sobre un grafo $G = (V, E)$ será $O(|V| + |E|)$.

3.6.2. Búsqueda en Amplitud

Esta búsqueda, también conocida como BFS por sus siglas en inglés (breadth first search), tiene aplicaciones similares a la búsqueda en profundidad: En primera instancia, se ocupa de visitar todos los vértices alcanzables desde algún vértice v, al que llamaremos fuente. Luego, realiza una búsqueda que podríamos describir como una búsqueda en capas: Primero, visita todos los vecinos de la fuente (capa 1), después visita a todos los vértices no visitados que son vecinos de aquéllos en la capa 1 (capa 2). El proceso se repite hasta que ya no haya más vértices que visitar, o lo que es lo mismo, hasta que no tengamos más capas.

Este algoritmo suele hacer uso de una cola Q como estructura de datos con el fin de conocer cuál es el siguiente vértice por visitar.

A continuación se muestra el algoritmo de esta búsqueda:

Algoritmo Búsqueda en Amplitud

```
1: función PREPROCESO(V, fuente)
2:     para cada v ∈ V hacer
3:         estado(v) ← no visitado
4:         l(v) ← ∞
5:     fin para
6: fin función
7: u ← fuente
8: Q.ingresar(u)
9: estado(u) ← visitado
10: l(u) = 0
11: función BÚSQUEDA EN AMPLITUD
12:     mientras Q ≠ ∅ hacer
13:         v ← Q.siguiente
14:         para cada w ∈ N(v) hacer
15:             si estado(w) = no visitado entonces
16:                 Q.ingresar(w)
17:                 estado(w) ← visitado
18:                 l(w) ← l(v) + 1
19:             fin si
20:         fin para
21:     fin mientras
22: fin función
```

Es posible que este proceso te recuerde al algoritmo de Dijkstra. Esto es porque la búsqueda en amplitud es la técnica de Dijkstra modificada; en este caso no hacemos uso de pesos, sino de la cantidad de aristas (es la razón por la que únicamente se puede sumar 1 en la línea 18 del algoritmo).

Este método no es recursivo como lo es la búsqueda en profundidad, sin embargo también requiere la visita de todos los vértices y de todas las aristas, por lo que su complejidad sobre un grafo $G = (V, E)$ es también $O(|V| + |E|)$.

3.6.3. Aplicaciones de Búsquedas

Podemos hacer uso de ambas búsquedas para verificar algunas propiedades en grafos, entre ellas saber si el grafo es conexo, contar el número de componentes, verificar si el grafo es bipartito, encontrar la ruta más corta entre dos vértices y las aristas y vértices de corte, entre otras cosas. En este apartado discutiremos dichos algoritmos.

Verificar si un Grafo es Conexo

Basta con ejecutar cualquiera de los dos algoritmos. Si al final todos los vértices fueron visitados, entonces el grafo es conexo, lo que también se puede ver como un grafo de un único componente. Cuando, en lugar de eso, requerimos saber cuántos componentes tiene un grafo, utilizamos el método siguiente.

Componentes de un Grafo

Para encontrar el número de componentes, podemos realizar repetidamente cualquiera de los dos algoritmos de búsquedas vistos cada vez que se encuentre un vértice no visitado, ya que si un vértice no ha sido visitado después de hacer alguna búsqueda, sólo puede significar que pertenece a otro componente.

El algoritmo que hace uso de una búsqueda en profundidad se muestra a continuación. Si quisiéramos hacer uso de una búsqueda en amplitud, únicamente tendríamos que llamar a esa función.

Algoritmo Contar componentes

```
1: componentes ← 0
2: para cada v ∈ V hacer
3:     si estado(v) = no visitado entonces
4:         Búsqueda en profundidad(v)
5:         componentes ← componentes + 1
6:     fin si
7: fin para
```

Verificar si un Grafo es Bipartito

Una buena idea para saber si un grafo es bipartito es utilizar una técnica conocida como *coloración*, y consiste en "colorear" los vértices de una manera correcta: Sean X y Y las particiones de los vértices del grafo. Sabemos que si $x \in X$, entonces $N(x) \subset Y$, lo que en el plano de colores significa que si x es de color X entonces sus vecinos son de color Y; análogamente, para los vértices $y \in Y$. De esta forma, si realizamos una búsqueda en amplitud o en profundidad recordando el color del predecesor del vértice actual, podremos determinar si estos pertenecen a la misma partición.

A continuación se muestra la modificación del algoritmo de búsqueda en amplitud para verificar si el grafo es bipartito; a diferencia de la búsqueda en amplitud regular, aquí tendremos tres estados: -1 (no visitado), 0 (color de X) y 1 (color de Y). Si existen dos "capas" consecutivas con el mismo color de vértices, entonces el grafo no es bipartito.

Algoritmo Bipartita en Amplitud

```
1: Q: cola
2: función PREPROCESO(V, fuente)
3:     para cada v ∈ V hacer
4:         color(v) ← −1
5:     fin para
6:     u ← fuente
7:     Q.ingresar(u)
8:     color(u) = 1
9: fin función
10: función BIPARTITA EN AMPLITUD
11:     mientras Q ≠ ∅ hacer
12:         v ← Q.siguiente
13:         para cada u ∈ N(v) hacer
14:             si color(u) = −1 entonces
15:                 Q.ingresar(u)
16:                 color(u) ← ¬color(v)
17:             si no si color(u) = color(v) entonces
18:                 regresar: No es bipartita
19:             fin si
20:         fin para
21:     fin mientras
22:     regresar: Es bipartita
23: fin función
```

Ruta más Corta en un Grafo Simple

Cuando vimos el algoritmo de Dijkstra estudiamos que éste es útil para encontrar la ruta más corta a partir de un vértice en un grafo con pesos. En este caso, haremos algo similar, pero no nos fijaremos en sus pesos, sino en la cantidad de aristas que se

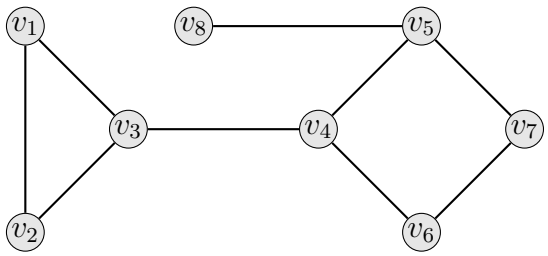
cuentan a partir de un vértice, esto, utilizando una búsqueda en amplitud.

Necesitaremos hacer uso de una cola para obtener el siguiente vértice u, para el cual actualizaremos $l(v)$ tal que $v \in N(u)$, y lo hacemos en tiempo constante (propiedad de las colas), y dado que $w(u, v) = 1$, la cola hace el trabajo de encontrar al vértice con la $l(u)$ mínima.

Aristas y Vértices de Corte

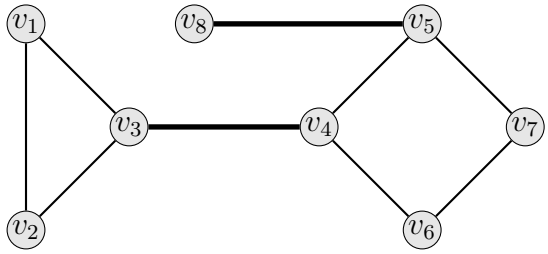
Si se retira de un grafo una arista uv y este movimiento provoca la separación del grafo en más componentes, la arista uv es conocida como *arista de corte*. En particular, en el caso de grafos clasificados como árboles, podemos garantizar que todas las aristas que lo conforman son de corte apoyándonos en el teorema 20.

Ejemplo 129. Encuentra las aristas de corte del siguiente grafo.



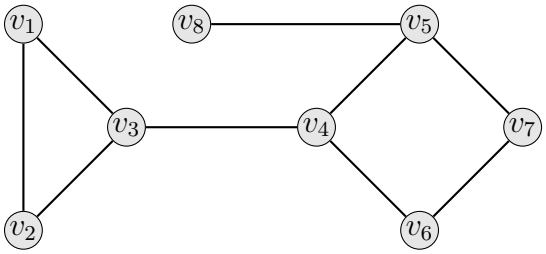
Solución. Es fácil ver que v_3v_4 es una arista de corte, pues si se omite, resultarán los subgrafos conformados por $\{v_1, v_2, v_3\}$ y $\{v_4, v_5, v_6, v_7, v_8\}$. Otra arista de corte es v_5v_8, que permite la obtención del subgrafo trivial v_8 y el subgrafo con los vértices $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$.

Marcando las aristas de corte:

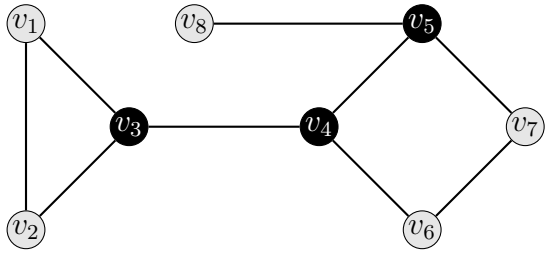


Después de abordar la definición de arista de corte, la definición de vértice de corte resulta intuitiva, de cualquier forma la mencionaremos: Un vértice de corte es aquél que puede particionar una gráfica G en dos subgrafos no nulos G_1, G_2, dicho de otra forma, un vértice de corte es lo único que tienen en común G_1 y G_2.

Ejemplo 130. Encuentra los vértices de corte del siguiente grafo.



Solución.



Una forma para encontrar las aristas y los vértices de corte es probar, para cada elemento de la gráfica, si su eliminación resulta en un mayor número de componentes

en el grafo dado haciendo uso de alguna de las búsquedas que vimos. No obstante, en [18] se presenta un método más eficiente para resolver este tipo de problemas.

4

Técnicas Avanzadas para la Resolución de Problemas

A lo largo de este capítulo retomaremos varios de los temas vistos previamente en *Análisis Combinatorio*, *Teoría de Números* y *Teoría de Grafos*, y añadiremos un grado de dificultad: el tamaño de los datos con los que se trabaja.

Así pues, en este capítulo se evidenciará, a través de varios ejemplos, la integración de la teoría matemática antes vista y varias técnicas avanzadas en el ámbito de la programación competitiva.

Pese a la teoría presentada, tanto en los capítulos anteriores como en éste, no es posible teorizar una fórmula que nos diga cuál es el método correcto para resolver un problema, e incluso puede suceder que haya más de una manera eficiente de hacerlo; ese aprendizaje se adquiere a través de la práctica constante, algo que, invariablemente, estará en manos únicamente del lector.

4.1. Búsqueda Completa

Como su nombre lo dice, la búsqueda completa, o exhaustiva, es una técnica que se basa en recorrer todo el espacio de búsqueda para encontrar la solución; este tipo de búsquedas pueden mejorarse en algunas ocasiones acortando el espacio a través del establecimiento de condiciones, sin embargo, esto sucederá en situaciones muy

específicas, en donde sea claro que un subconjunto del espacio de búsqueda es excluible de la solución al problema planteado.

Cuando se recurre a una búsqueda completa no es posible obtener *Wrong Answer* como veredicto (si el método fue correctamente aplicado), porque esta técnica revisa todas las posibles respuestas, no obstante, corremos el riesgo de recibir un *Time Limit*, por lo que antes de realizar una búsqueda completa es necesario hacer un análisis de complejidad; obtenida la complejidad del algoritmo propuesto, así como el tamaño de la entrada, es posible determinar si una búsqueda completa es o no una opción de solución viable. En ocasiones, se te presentarán problemas donde no hay otra solución aparente, o si la hay, parece muy sobrada para los quizá pequeños casos de prueba. Mientras nuestra solución cumpla con las restricciones de tiempo y memoria, tendremos en las manos una buena idea aunque ésta sea una sencilla búsqueda completa.

Como ya hemos mencionado, si el algoritmo rebasara el tiempo permitido, una buena idea sería acotar el espacio de búsqueda por medio de alguna característica establecida, pero si esto no fuera suficiente, tal vez el algoritmo que solucione el problema en cuestión no es una búsqueda completa.

Ejemplo 131. Considera un conjunto dado de 30 enteros distintos. ¿Cuántos subconjuntos de tamaño 5 se pueden formar cuya suma tenga la forma $7k + 3$, para $k \in \mathbb{Z}$?

> **Solución.** En este caso, tendremos que recurrir a una búsqueda completa, en donde el espacio de búsqueda serán todos los posibles subconjuntos de tamaño 5 del conjunto de números proporcionado en la entrada.
>
> ¿Por qué la búsqueda completa parece una buena idea para solucionar este problema? Pese a que buscamos la cantidad de subconjuntos que cumplen con la condición pedida, no tenemos una forma de decidir si un subconjunto suma $7k + 3$ además de verificar si eso es cierto. Por otro lado, la complejidad del algoritmo que se presenta como solución es únicamente $\binom{30}{5}$, pues estamos obteniendo todos los posibles subconjuntos de tamaño 5 (a pesar de que hay cinco ciclos anidados, estos no recorren todos los elementos del vector, sino únicamente los necesarios para cons-

truir subconjuntos distintos a los previos).

```
int solucion()
{
    int i, numero, validos = 0, i1, i2, i3, i4, i5;
    vector<int> enteros;
    for (i = 0; i < 30; i = i + 1)
    {
        cin>>numero;
        enteros.push_back(numero);
    }

    for (i1 = 0; i1 < 30; i1 = i1 + 1)
        for (i2 = i1 + 1; i2 < 30; i2 = i2 + 1)
            for (i3 = i2 + 1; i3 < 30; i3 = i3 + 1)
                for (i4 = i3 + 1; i4 < 30; i4 = i4 + 1)
                    for (i5 = i4 + 1; i5 < 30; i5 = i5 + 1)
                        if ( ( enteros[i1] + enteros[i2] + enteros[i3]
                        + enteros[i4] + enteros[i5] ) % 7 == 3 )
                            validos = validos + 1;

    return validos;
}
```

Las búsquedas en amplitud y en profundidad son también claros ejemplos de búsquedas completas, y podrás leer más del tema en el capítulo de *Teoría de Grafos*. No obstante, por ahora abordaremos un ejemplo relacionado con las permutaciones de un arreglo.

4.1.1. Next Permutation

En el capítulo de *Análisis Combinatorio* estudiamos la manera de contar los posibles acomodos de un arreglo de objetos sin tener que obtener cada configuración distinta, por desgracia, en ocasiones, más que saber la cardinalidad de dicho conjunto, requerimos el conjunto en sí. Para nuestra fortuna, en C++ existe ya una función

para obtenerlo.

La función de nombre *Next Permutation* (alojada en $<algorithm>$), reordena los elementos de cierto arreglo A en el intervalo $[inicio, fin)$ de tal manera que obtiene siempre la permutación inmediatamente mayor a la actual en orden lexicográfico ascendente.

La forma de la función es $next\_permutation(A + inicio, A + fin)$, y devuelve *true* si es posible obtener la siguiente permutación; si no es posible (porque la permutación actual del arreglo o del vector es la mayor), devuelve *false*. Por ejemplo, dado el arreglo $A =< a, b, c, d, e >$, si aplicamos la sentencia $next\_permutation(A + 1, A + 3)$, obtendremos todas las posibles permutaciones del subarreglo $[1, 3) = [1, 2]$, esto es $< c, b >$, pues $< b, c >$ era la configuración inicial y ésta no será obtenida. Si, por otro lado, buscamos las permutaciones de todo el arreglo A, la sentencia correcta será $next\_permutation(A + 0, A + 5) = next\_permutation(A, A + 5)$.

Ejemplo 132. Resuelve el ejemplo 120 (visto en *Teoría de Grafos*) haciendo uso de la función *Next Permutation*.

> **Solución.** Recordando, este problema requería encontrar las rutas óptimas de varias secciones de caminos; en este camino estaban involucrados tres puntos p_1, p_2, p_3, para los que debíamos encontrar todas las maneras de pasar por ellos. De esta forma, elegiríamos la ruta óptima de A a algún p_i, la ruta óptima para recorrer los tres p_j y la ruta óptima para ir de determinado p_k a B.
>
> En el ejemplo 120 tuvimos que encontrar "a mano" todas las posibles maneras de pasar por p_1, p_2 y p_3. En este caso únicamente llamaremos a la función *Next Permutation*, que se encargará de obtener todas las permutaciones, convirtiendo seis líneas de código en únicamente una; el resto de la solución seguirá el mismo procedimiento que la antes vista.

```
1  int solucion(vector<vector<int>> &M)
2  {
3      int A, B, p1, p2, p3, r_min = INF;
4      cin>>A>>B;
5      cin>>p1>>p2>>p3;
```

```
6       vector<int> puntos({p1,p2,p3});
7
8       FloydWarshall(M);
9
10      do
11      {
12          r_min = min( r_min, M[A][puntos[0]] + M[puntos[0]][puntos[1]]
13                + M[puntos[1]][puntos[2]] + M[puntos[2]][B] );
14      }while (next_permutation(puntos, puntos + 3));
15
16      return r_min;
17  }
```

Ejemplo 133. La entrada de este problema será una cadena binaria de, a lo más, 20 dígitos. Si dichos dígitos pueden ser cambiados de lugar (pero no modificado su valor), ¿cuántas cadenas se pueden formar, partiendo de la cadena original, que sean múltiplos de 79?

> **Solución.** Al poder cambiar de lugar los dígitos, lo que estamos buscando son todas las posibles permutaciones de la cadena original, para luego convertirlo a un entero en base 10 y revisar si 79 lo divide.

```
1   int solucion()
2   {
3       int divisibles = 0;
4       string cadena;
5       cin>>cadena;
6
7       //Damos una configuración inicial a la cadena: la menor posible.
8       sort(cadena.begin(), cadena.end());
9
10      do
11      {
12          bitset<20> numero(cadena);
13          if (numero.to_ulong() % 79 == 0)
14              divisibles = divisibles + 1;
15      }while( next_permutation(cadena.begin(), cadena.end()) )
16
17      return divisibles;
18  }
```

A lo largo de esta sección hemos visto ejemplos específicos de búsquedas completas, y mencionamos también que las búsquedas en amplitud y en profundidad, realizadas sobre un grafo, forman parte de esta categoría de solución. No obstante, esto no implica que todas las búsquedas completas deban resolverse con alguno de los tópicos vistos aquí. En tanto recorras todo el espacio de búsqueda, sin importar de qué manera lo hagas, estarás tratando con una búsqueda completa.

4.2. Divide y Vencerás

Esta técnica consiste en resolver un problema haciéndolo más "simple" al dividirlo en subproblemas. Podemos decir que consta de los siguientes aspectos:

- Dividir el problema principal en dos o más subproblemas.
- Resolver cada subproblema, lo que usualmente se lleva a cabo de manera recursiva.
- Combinar las soluciones de los subproblemas para construir la solución del problema principal.

Para comprender mejor el concepto, abordaremos lo que se conoce como *búsqueda binaria* a continuación.

4.2.1. Búsqueda Binaria

Si queremos localizar cierto dato k dentro de un arreglo dado de longitud n, quizá la manera natural de buscarlo es de manera lineal, esto es, comenzar en el primer elemento del arreglo y verificar si es nuestro dato, si no lo es, continuar con la verificación del segundo elemento, luego del tercero y así sucesivamente hasta llegar al último. Claro está que el peor caso se presenta cuando el dato buscado se encuentra en la n-ésima posición del arreglo, y por tanto, la complejidad de dicha búsqueda

será $O(n)$. No obstante, buscar ese mismo dato con un método binario resulta en una complejidad de $O(log_2 n)$.

Para que una búsqueda binaria pueda ser ejecutada, la primera condición es que los elementos del arreglo hayan sido ordenados de manera ascendente (el ordenamiento también se puede realizar de manera descendente siempre que se consideren los cambios necesarios para el resto del algoritmo).

El algoritmo funciona así: Partimos el arreglo por la mitad y comparamos a k con el elemento que particionó al arreglo, con lo que pueden ocurrir tres escenarios:

- Si $k < mitad$, entonces la búsqueda debe proseguir en la primera mitad del arreglo.
- Si $k > mitad$, entonces la búsqueda debe proseguir en la segunda mitad del arreglo.
- Si $k = mitad$, habremos terminado.

Si ocurre cualquiera de los dos primeros puntos, entonces el algoritmo continuará exactamente de la misma forma, pero ahora con la mitad en la que sabemos se encuentra k (si es que k realmente pertenece al arreglo). De esta forma, buscaremos primero en una mitad del arreglo, luego en una cuarta parte, en un octavo y así sucesivamente, hasta que nuestro espacio de búsqueda se reduzca a un único elemento, de ahí su complejidad logarítmica.

Veremos el funcionamiento detallado de una búsqueda binaria en el ejemplo que se presenta a continuación.

Ejemplo 134. Encuentra los datos 3 y 14 en el siguiente arreglo haciendo uso de una búsqueda binaria.

$$< 9, 16, -9, 15, 12, 6, 3, -8 >$$

Solución. Comencemos por ordenar los datos.

$$< -9, -8, 3, 6, 9, 12, 15, 16 >$$

Hecho esto, podemos aplicar la búsqueda binaria; buscaremos primero al número 3: Partiendo el arreglo a la mitad, obtendremos los subarreglos:

$$< -9, -8, 3, 6 > \qquad \text{y} \qquad < 9, 12, 15, 16 >$$

$3 < 6$, por lo que descartaremos el segundo subarreglo y nos enfocaremos en el primero, que a su vez, se partirá en:

$$< -9, -8 > \qquad \text{y} \qquad < 3, 6 >$$

$3 > -8$, con lo que sabemos que el elemento buscado está en el segundo subarreglo. Finalmente:

$$< 3 > \qquad \text{y} \qquad < 6 >$$

Y $3 = 3$, terminando allí la búsqueda.

Ahora buscaremos a 14. Cuando partimos por primera vez al arreglo, obtenemos nuevamente $< -9, -8, 3, 6 >$ y $< 9, 12, 15, 16 >$, pero en esta ocasión deberemos ir por el segundo subarreglo, pues $14 > 6$. Luego, tendremos $< 9, 12 >$ y $< 15, 16 >$, en donde, nuevamente, elegiremos el segundo subarreglo. Finalmente, veremos $< 15 >$ y $< 16 >$, y deberíamos escoger el primer subconjunto, pero este ya es de tamaño 1 y no es igual a 14, con lo que concluimos que 14 no está presente en el arreglo.

Ejemplo 135. Dados dos arreglos de números del mismo tamaño, encuentra el subarreglo más grande que tengan en común.

Solución. Para este problema requeriremos dos cosas: La primera será, obviamente, una búsqueda binaria, y la segunda será una implementación de *rolling hash* (que encontrarás en el capítulo de *Teoría de Números*); para la realización de la búsqueda, haremos uso de una función conocida como *"lower bound"*, dentro de la cual no profundizaremos en esta obra, pero que puedes encontrar en [22].

Denominaremos *inicio* al primer número de cada arreglo y *fin* al último. Luego, fijaremos un $k_1 = \lfloor (inicio + fin)/2 \rfloor$ (para encontrar la mitad del arreglo); aplicaremos un procedimiento de *rolling hash* a cada arreglo para obtener el equivalente de cada subarreglo de tamaño k_1, y compararemos. En este caso, y para ser estrictos, estamos usando la función *piso*

al obtener k_1, sin embargo, en el programa es común que simplemente se obtenga de $(inicio + fin)/2$ y se guarde en un entero, de esta forma se obtendría el mayor entero menor o igual que la respuesta, que es, de hecho, la definición de la función piso.

Después existen dos opciones: Que exista un subarreglo tamaño k_1 en común, o no.

Si existe tal subarreglo, es probable que exista uno de mayor tamaño. Para verificarlo, repetiremos el proceso, pero nuestro nuevo *inicio* será $k_1 + 1$ y el fin permanecerá de la misma manera. Luego, nombraremos un $k_2 = \lfloor (inicio + fin)/2 \rfloor$, y preguntaremos si se cumple la igualdad entre cada par de subarreglos de longitud k_2.

Si no existe, habrá que buscar subarreglos de menor tamaño que k_1, y fijaremos al fin como $k_1 - 1$ conservando el inicio original. Buscaremos así la coincidencia de los subarreglos tamaño k_2, donde $k_2 = \lfloor (inicio + fin)/2 \rfloor$.

Este procedimiento se repetirá hasta que el *inicio* coincida con el fin, y el k_i que resulte en esa iteración será el tamaño del máximo subarreglo en común entre los arreglos de la entrada.

```
1  int n; //Tamaño de los arreglos.
2  vector<int> arr1, arr2;
3  set<pair<int, int>> miset;
4
5  //Bases y módulos para aplicación de rolling hash.
6  int B1 = 29, B2 = 31, mod1 = 1000000033, mod2 = 1000000021;
7
8  //Busca si NO tienen en común un subarreglo de tamaño k.
9  bool no_existe(int k)
10 {
11     int i;
12     long long B1_pot, B2_pot, h1, h2;
13
14     miset.clear();
15     B1_pot = ExpRapida_modulo(B1, k - 1, mod1);
16     B2_pot = ExpRapida_modulo(B2, k - 1, mod2);
17
```

```
18      //PRIMER ARREGLO.
19      h1 = obtener_hash(arr1, k, B1, mod1);
20      h2 = obtener_hash(arr1, k, B2, mod2);
21
22      //Queremos saber si existe.
23      miset.insert({h1, h2});
24
25      for (i = k; i < n; i = i + 1)
26      {
27          h1 = ( (h1 - arr1[i - k] * B1_pot % mod1 + mod1) * B1 % mod1
28              + arr1[i]) % mod1;
29          h2 = ( (h2 - arr1[i - k] * B2_pot % mod2 + mod2) * B2 % mod2
30              + arr1[i]) % mod2;
31          miset.insert({h1, h2});
32      }
33
34      //Buscar en el SEGUNDO ARREGLO.
35      h1 = obtener_hash(arr2, k, B1, mod1);
36      h2 = obtener_hash(arr2, k, B2, mod2);
37
38      bool no_existe = true;
39      if ( miset.find({h1, h2}) != miset.end() )
40          no_existe = false;
41
42      for (int i = k; i < n and no_existe; i = i + 1)
43      {
44          h1 = ( (h1 - arr2[i - k] * B1_pot % mod1 + mod1) * B1 % mod1
45              + arr2[i]) % mod1;
46          h2 = ( (h2 - arr2[i - k] * B2_pot % mod2 + mod2) * B2 % mod2
47              + arr2[i]) % mod2;
48
49          if ( miset.find({h1, h2}) != miset.end() )
50              no_existe = false;
51      }
52
53      return no_existe;
54  }
55
56  //El tamaño del arreglo más pequeño que NO tienen en común.
57  int lower_bound (int primero, int ultimo)
58  {
59      int val = 1;
60      int tam;
61      int cuenta, salto;
62
63      cuenta = ultimo - primero;
```

```
64      while (cuenta > 0)
65      {
66          tam = primero;
67          salto = cuenta / 2;
68          tam = tam + salto;
69
70          if (no_existe(tam) < val)
71          {
72              primero = tam = tam + 1;
73              cuenta = cuenta - salto - 1;
74          }
75          else cuenta = salto;
76      }
77      return primero;
78  }
79
80  void solucion()
81  {
82      int i, respuesta;
83      cin>>n;
84      arr1.resize(n);
85      arr2.resize(n);
86
87      for ( i = 0; i < n; i = i + 1)
88          cin>>arr1[i];
89
90      for ( i = 0; i < n; i = i + 1)
91          cin>>arr2[i];
92
93      //Búsqueda en el intervalo [1, n + 1).
94      respuesta = lower_bound(1, n + 1) - 1;
95      cout<<respuesta<<"\n";
96  }
```

El ejemplo anterior muestra la importancia de la comprensión de una búsqueda binaria, sin embargo ya existe una búsqueda binaria implementada dentro de la librería $< algorithm >$, cuyo uso se muestra en el ejemplo a continuación.

Ejemplo 136. Dada una lista de enteros y algún número s, determina si existe un par de elementos en la lista cuya suma sea igual a s. (Problema "Esto es fácil", tomado de [3]).

Solución. Llamemos a y b al par de enteros que sumen s. Podemos decir entonces que $a = s - b$. Esta relación nos da la pauta para entender que, para cada b, habrá que buscar si existe el número $s - b$ dentro de la lista, y para verificarlo utilizaremos un procedimiento de búsqueda binaria.

```
bool solucion()
{
    int s, a, b;
    vector<int> numeros;
    cin>>s;

    while (cin>>n)
        numeros.push_back(n);

    sort(numeros.begin(), numeros.end());
    for (auto b: numeros)
    {
        a = s - b;
        if (binary_search(arreglo_enteros.begin(),
           arreglo_enteros.end(), a))
            return true;
    }

    return false;
}
```

4.2.2. Bisección

Con lo visto previamente, resulta claro que una búsqueda binaria se podrá aplicar a un conjunto discreto de datos, es decir, un conjunto contable (como lo es un arreglo o un vector). Pues bien, el método de la Bisección resulta muy similar a una búsqueda binaria, con la diferencia de que una bisección se podrá aplicar a un intervalo continuo. Con el siguiente ejemplo será más claro el funcionamiento del método.

Ejemplo 137. Encuentra la raíz cuadrada de 7.

Solución. Realizaremos la búsqueda de la raíz de nuestro número haciendo aproximaciones con un método muy parecido al de una búsqueda binaria, y las evaluaremos elevándolas al cuadrado buscando un resultado cercano a 7.

Ya que 7 es primo, su raíz cuadrada no es racional, sin embargo sí podemos dar una buena aproximación.

Comencemos probando el valor $7/2 = 3.5$:

$$(3.5)^2 = 12.25$$

Ya que el número obtenido es mayor que 7, buscaremos en la primera mitad del intervalo $[0, 7]$, esto es, en $[0, 3.5]$, para el que verificaremos el punto medio $(3.5 + 0)/2 = 1.75$:

$$(1.75)^2 = 3.0625$$

Lo anterior nos dice que la raíz de 7 es un número menor que 3.5, pero mayor que 1.75. Probemos entonces el punto medio de $[1.75, 3.5]$, es decir, $(3.5 + 1.75)/2 = 5.25/2 = 2.625$:

$$(2.625)^2 = 6.890625$$

La anterior ya es una buena aproximación, pues $|7-6.890625| = 0.109375$, sin embargo, buscamos una mayor exactitud, para lo que trataremos con un número ligeramente mayor que 2.625, por lo que ahora analizaremos el punto medio del intervalo [2.625, 3.5], 3.0625:

$$(3.0625)^2 = 9.37890625$$

Desafortunadamente, este número nos aleja de 7, así que habrá que intentar con uno menor, específicamente, con el punto medio de [2.625, 3.0625], 2.84375:

$$(2.84375)^2 = 8.0869140625$$

Seguimos por arriba de 7, así que ahora iremos por $(2.625 + 2.84375)/2 = 2.734375$:

$$(2.734375)^2 = 7.476806640625$$

Nuevamente, buscamos un número menor. Es el turno de $(2.625 + 2.734375)/2 = 2.6796875$:

$$(2.6796875)^2 = 7.18072509765625$$

Seguimos reduciéndolo con $(2.625 + 2.6796875)/2 = 2.65234375$:

$$(2.65234375)^2 = 7.0349273681640625$$

Tenemos una mejor aproximación, pero aún la podemos acercar más al valor real con $(2.625 + 2.65234375)/2 = 2.638671875$:

$$(2.638671875)^2 = 6.962589263916$$

Probamos con una aproximación un poco mayor con $(2.638671875 + 2.65234375)/2 = 2.6455078125$:

$$(2.6455078125)^2 = 6.998711585998535$$

Ahora con $(2.6455078125 + 2.65234375)/2 = 2.64892578125$:

$$(2.64892578125)^2 = 7.016807794570923$$

Para $(2.6455078125 + 2.64892578125)/2 = 2.647216796875$:

$$(2.647216796875)^2 = 7.007756769657135$$

Con $(2.6455078125 + 2.647216796875)/2 = 2.6463623046875$:

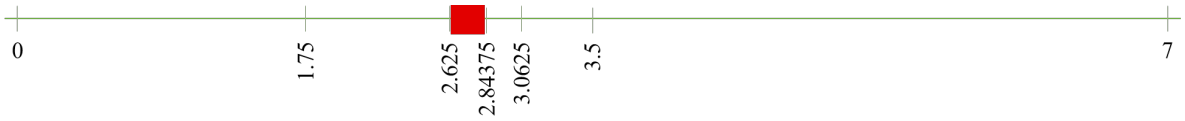
$$(2.6463623046875)^2 = 7.003233447670937$$

Reduciendo un poco más,
$(2.6455078125 + 2.6463623046875)/2 = 2.64593505859375$:

$$(2.64593505859375)^2 = 7.000972334295511$$

En esta ocasión, nos conformaremos con esta exactitud, terminando las iteraciones aquí y quedándonos con 2.64593505859375 como la raíz de 7.

Explicando un poco mejor lo que hicimos arriba, comenzaremos por el hecho de que nuestro primer intervalo de búsqueda fue $[0, 7]$, y "partirlo" a la mitad nos permitió trabajar sobre un espacio más reducido (el intervalo $[0, 3{,}5]$), con lo que descartamos completamente los reales mayores que 3.5. Probando el valor de 3.5, nos damos cuenta de que la raíz de 7 es menor que 3.5, así que probamos con el punto medio entre 0 y 3.5: 1.75. Nuevamente, sustituyendo la raíz por 1.75 vemos que esta vez necesitamos un valor mayor que 1.75, pero menor que 3.5, probando con el punto medio de 1.75 y 3.5: 2.625. Otra vez requerimos un número mayor que 2.625, pero menor que 3.5: 3.0625, para luego buscar el punto medio de 2.625 y 0.0623: 2.84375.



En la imagen de arriba no hay espacio suficiente para mostrar todas las bisecciones que realizamos, sin embargo, en el ejemplo 137 repetimos este procedimiento hasta encontrar un valor que nos permitiera obtener la menor diferencia entre lo que deberíamos obtener y lo que obtenemos, en este caso, entre 7 y 7.000972334295511, respectivamente.

Esta diferencia es conocida como *margen de error*, y en términos prácticos, podemos decir que es lo que nos permitiremos fallar en la exactitud, es decir, el valor absoluto de la diferencia entre lo que obtenemos y lo que deberíamos obtener.

Este procedimiento tiene dos métodos de paro: Uno de ellos consiste en establecer previamente cuántas iteraciones se deben llevar a cabo, y el otro es justamente hallar el primer valor en el que se alcance o se mejore el margen de error.

Ejemplo 138. Utiliza el método de bisección para encontrar la raíz de la expresión $e^x + x^3 - 2x^2 = 0$ en cualquier intervalo dado $[a, b]$ después de realizar 10 iteraciones. En caso de que no exista un valor, imprime X.

Solución. Aplicando la teoría vista previamente, será sencillo codificar la solución.

```
double f(double x)
{
    return exp(x) + x * x * x - 2 * x * x;
}

void solucion()
{
    double a, b, c;
    int iteraciones = 10;
    bool sol_encontrada = false;
    cin>>a>>b;

    while (iteraciones--)
    {
        c = (a + b) / 2;

        //En caso de que se encuentre la raíz exacta.
        if (fabs( f(c) ) <= 0.001)
        {
            sol_encontrada = true;
            break;
        }

        //Encontrar el siguiente punto medio.
        if (f(c) * f(a) < 0)
            b = c;
        else
            a = c;
    }

    if (sol_encontrada)
        cout<<c;
    else
        cout<<"X";
}
```

4.2.3. Ordenamiento de Merge

Es posible que conozcas varios métodos de ordenamiento, y todos ellos tienen complejidades distintas. Esta técnica en particular, se caracteriza por ejecutarse en tiempo $O(n \cdot log\ n)$, donde n es la cantidad de elementos a ordenar, y se basa en el principio de Divide y Vencerás.

Pensemos en introducir nuestros datos a un arreglo o a un vector. El algoritmo de Merge Sort divide por la mitad a dicho arreglo obteniendo con ello dos subarreglos, y con estos hace lo mismo hasta obtener únicamente arreglos de tamaño 1, que, obviamente, ya están ordenados. Luego, ordena los bloques que resultan de unir pares de arreglos de cardinalidad 1, para después ordenar entre sí los bloques que surgen del movimiento anterior. Eventualmente, obtendremos acomodos de tamaño $n/2$, que, al ensamblar, nos otorgarán el arreglo entero.

Veamos el algoritmo a través de un ejemplo.

Ejemplo 139. Utiliza el algoritmo de Merge Sort para ordenar el siguiente arreglo:

$$< 13, 2, -1, 40, 8, -12, 201, 40, 31, 0 >$$

Solución. Nuestro arreglo es de tamaño 10. Dividiéndolo por la mitad, obtendremos los siguientes arreglos, a los que, por comodidad, les nombraremos "el segundo nivel":

$$< 13, 2, -1, 40, 8 > \qquad < -12, 201, 40, 31, 0 >$$

Dividimos nuevamente a la mitad (en lo posible) para obtener el tercer nivel:

$$< 13, 2, -1 > \quad < 40, 8 > \quad < -12, 201, 40 > \quad < 31, 0 >$$

El cuarto nivel será:

$$< 13, 2 > \ < -1 > \ < 40 > \ < 8 > \ < -12, 201 > \ < 40 > \ < 31 > \ < 0 >$$

Los únicos arreglos que aún no alcanzan tamaño 1 son el primero y el quinto, por lo que son los únicos que pasarán al quinto nivel:

$< 13 >$ $< 2 >$ $< -12 >$ $< 201 >$

Obtenidos ya los arreglos de tamaño 1, vamos "a la inversa" para ordenarlos al tiempo que los unimos; todo arreglo de tamaño 1 está ordenado, por lo que comenzaremos por concatenar el quinto nivel:

$< 2, 13 >$ $< -12, 201 >$

Uniendo y ordenando los arreglos del cuarto nivel por pares, y considerando los cambios en el quinto:

$< -1, 2, 13 >$ $< 8, 40 >$ $< -12, 40, 201 >$ $< 0, 31 >$

Trabajando en el tercer nivel:

$< -1, 2, 8, 13, 40 >$ $< -12, 0, 31, 40, 201 >$

Para finalmente unir y ordenar los arreglos anteriores y obtener el arreglo de tamaño 10 (nivel 1) ordenado:

$$< -12, -1, 0, 2, 8, 13, 31, 40, 40, 201 >$$

Ejemplo 140. Codifica el ordenamiento de Merge.

Solución.

```
1  vector<int> MergeSort(vector<int> arreglo)
2  {
3      if (arreglo.size() == 1)
4          return arreglo;
5
6      //División de arreglos.
7      vector<int> v1(arreglo.begin(), arreglo.begin()+arreglo.size()/2);
8      vector<int> v2(arreglo.begin()+arreglo.size()/2, arreglo.end());
9
10     vector<int> A1 = MergeSort(v1);
11     vector<int> A2 = MergeSort(v2);
12
13     return Combinar(A1, A2);
```

```
14  }
15
16  //Se ordenan los subarreglos.
17  vector<int> Combinar(vector<int> A1, vector<int> A2)
18  {
19      vector<int> respuesta;
20      int i = 0, j = 0;
21
22      //Unión de subarreglos ordenándolos.
23      while (i < A1.size() and j < A2.size())
24      {
25          if (A1[i] < A2[j])
26          {
27              respuesta.push_back(A1[i]);
28              i = i + 1;
29          }
30          else
31          {
32              respuesta.push_back(A2[j]);
33              j = j + 1;
34          }
35      }
36
37      //Inclusión de elementos que sobraron, ya sea en A1 o en A2.
38      if ( i == A1.size() )
39          for (; j < A2.size(); j = j + 1)
40              respuesta.push_back(A2[j]);
41      else
42          for (; i < A1.size(); i = i + 1)
43              respuesta.push_back(A1[i]);
44
45      return respuesta;
46  }
```

4.3. Greedy

Un problema se puede resolver de manera *greedy* si cumple las siguientes características:

- Subestructura óptima: La solución óptima contiene soluciones óptimas de subproblemas.
- Propiedad *greedy*: Si tomamos la opción que parece la mejor a cada paso, obtendremos la solución óptima. No es necesario considerar acciones pasadas ni futuras.

Por ejemplo, si tuviéramos únicamente las siguientes denominaciones de monedas: $\{25, 5, 1\}$, y quisiéramos saber cuál es la mínima cantidad de monedas necesarias para obtener exactamente cierta cantidad C, digamos 42, podemos elegir tantas monedas de las mayores denominaciones como se pueda, para lo que bastará elegir la mayor denominación menor o igual a la cantidad actual requerida, por tanto, escogemos primero una moneda de 25; restan $42 - 25 = 17$, para lo que hacemos el mismo procedimiento: elegimos una moneda de 5, y quedan $17 - 5 = 12$, después $12 - 5 = 7$, $7 - 5 = 2$, $2 - 1 = 1$ y $1 - 1 = 0$. La solución óptima contiene entonces soluciones óptimas de los subproblemas de 17, 12, 7, 2 y 1.

Otros ejemplos de algoritmos *greedy* son los algoritmos de Dijkstra, Prim-Jarníck y Kruskal (vistos en *Teoría de Grafos*) en los que, en cada iteración, se toma la mejor opción disponible para ese momento.

En ocasiones, ordenar los elementos dados puede ser de ayuda para identificar estrategias *greedy*, no obstante, es importante mencionar que existen problemas que, aunque tienen la propiedad de subestructura óptima, no es así con la propiedad *greedy*, por lo que hay que ser muy cuidadosos con el uso de esta técnica y hacer un análisis profundo del problema que se intenta resolver.

Ejemplo 141. Imagina que irás de viaje y recorrerás toda la república mexicana. Para ello, cada noche te hospedarás en un hotel distinto, cada uno con un precio diferente, pero todos ellos pertenecientes a la misma cadena. Esta cadena te brindará la ventaja de que, cada k noches, te otorgará una noche gratis, misma que puedes usar cuando tú prefieras. Encuentra la máxima cantidad de dinero que puedes ahorrar aprovechando el sistema de esta cadena hotelera considerando que en este momento no conoces la cantidad de noches que dormirás en los hoteles. (Problema tomado de [23]).

> **Solución.** No podemos nada más elegir los hoteles más caros para que estos sean los que no se cobren, pues no podemos asegurar que, para ese

momento, ya hayamos obtenido el número de regalos necesarios para cubrirlos; por ejemplo, en la situación en la que $k = 5$ y las primeras tres noches fueran las más caras (para ese momento no tendríamos ningún regalo disponible), o para un caso en que las tres noches más caras son la sexta, la séptima y la octava (para ese momento únicamente podemos cubrir una noche gratis).

Así pues, primero hay que asegurarnos de pagar k noches, y tomar como gratis la $(k+1)$-ésima; de hecho, bajo la misma lógica, diremos que todos los hoteles múltiplos de $k+1$ tendrán, "por default", costo 0. Esta técnica nos ayuda a asegurar que, al momento, es válido usar la noche gratis, pero no asegura que ahorremos la máxima cantidad de dinero, por lo que habrá que hacer más movimientos.

Pensemos en las noches en el intervalo $[k+2, 2k+1]$ (cuando existen). Si alguna de esas noches tiene un precio mayor que la $(k+1)$-ésima, entonces intercambiaremos la noche que elegimos no pagar. Una vez obtenida la segunda noche gratis (la noche $2k+2$), la tomaremos, por si no hay más noches, y analizaremos las noches del intervalo $[2k+3, 3k+2]$; si una de ellas cuesta más que la más barata del conjunto de las, hasta ahora, noches gratis, entonces intercambiaremos la noche que elegimos no pagar. Si después de hecho esto, en el intervalo $[2k+3, 3k+2]$ todavía hay una noche más cara que la más barata entre el conjunto de noches gratis, nuevamente intercambiaremos los elementos.

Continuaremos así con el proceso hasta terminar el viaje: Tomamos todas las noches de la forma $(k+1)n$ y las incluimos en el conjunto de noches gratis (por si no hay más noches), pero luego tendremos que comparar los precios con el resto de las noches que pueden ser tomadas e intercambiarlos si estos mejoran la cantidad de dinero que podemos ahorrar.

```
1  int solucion()
2  {
3      int k, i = 1, precio, ahorro = 0;
4      priority_queue<int, vector<int>, greater<int>> gratis;
5
6      cin>>k;
```

```
7       while (cin>>precio)
8       {
9           //Declaramos gratis todas las noches múltiplos de k+1.
10          if( i % (k + 1) == 0)
11              gratis.push(precio);
12
13          //Sustituimos si existen noches más caras.
14          else if(!gratis.empty() and gratis.top() < precio)
15          {
16              gratis.pop();
17              gratis.push(precio);
18          }
19
20          i = i + 1;
21      }
22
23      while(not gratis.empty())
24      {
25          ahorro = ahorro + gratis.top();
26          gratis.pop();
27      }
28
29      return ahorro;
30  }
```

4.3.1. Intervalos Disjuntos

El estilo de estos problemas incluye un intervalo dado $[a, b]$, que es necesario cubrir de subintervalos de tal forma que el número de subintervalos sea el máximo posible y que, a su vez, no se traslapen (sin importar que el intervalo no sea cubierto completamente). Estos subintervalos deben ser dados en la forma $[inicio, final)$.

Ejemplo 142. Una sala es usada para diferentes actividades en un día, cada una de las cuales tiene una hora de inicio y una hora de fin. Dado un conjunto de actividades en la forma $[hora\_inicio, hora\_final)$, encuentra la máxima cantidad de actividades que se pueden realizar en un día en la sala sin que se traslapen.

Solución. Para maximizar la cantidad de actividades en un día, hay que asegurarnos de desocupar la sala lo más pronto posible para así asignarla a otro evento. Así pues, ordenaremos las actividades en orden ascendente por la hora a la que terminan, e iremos tomando las actividades en ese mismo orden, con la única condición de que una actividad se puede tomar si su hora de inicio es mayor o igual que la hora de fin que el último evento tomado.

```
1  //Para ordenar por la hora de finalización.
2  bool f(pair<int, int> a, pair<int, int> b)
3  {
4      return a.second < b.second;
5  }
6
7  int solucion()
8  {
9      int n, i, cantidad = 0;
10     pair<int, int> ultima_actividad;
11     vector<pair<int, int>> actividades;
12     //First será la hora de inicio y second la de término.
13
14     cin>>n;
15     actividades.resize(n);
16     for (i = 0; i < n; i = i + 1)
17         cin>>actividades[i].first>>actividades[i].second;
18     sort(actividades.begin(), actividades.end(), f);
19
20     for (i = 0; i < n; i = i + 1)
21         if (actividades[i].first >= ultima_actividad.second)
22         {
23             cantidad = cantidad + 1;
24             ultima_actividad = actividades[i];
25         }
26
27     return cantidad;
28 }
```

4.3.2. Cobertura de Intervalos

Los problemas de este estilo establecen un intervalo $[a, b]$, y se requiere que dicho intervalo sea cubierto totalmente por el menor conjunto de subintervalos que cumplan ciertas características, mismos que podrán, o no, traslaparse, en contraste con la idea de intervalos disjuntos. Esos subintervalos también deben ser dados en la forma $[inicio, final)$.

Ejemplo 143. Se desea iluminar un túnel colocando lámparas en determinados puntos, cada una de las cuales tiene su propio rango de alcance $[inicio, final)$. Así dado un conjunto de lámparas, encuentra la cantidad mínima necesaria para iluminar todo el túnel.

> **Solución.** En este caso y dado que no nos importa que el alcance de las lámparas se traslape (es decir, que haya dos lámparas que coinciden al iluminar cierto espacio), ordenaremos de manera ascendente por el punto de inicio de cada rango, e iremos tomando las lámparas en ese orden; se debe cumplir que el alcance de cada lámpara comience antes de que la lámpara anterior termine, o bien, al mismo tiempo. Ahora bien, ya que requerimos que el número de lámparas sea el mínimo posible, el segundo parámetro de preferencia será que tenga el mayor entre los alcances disponibles en ese punto.
>
> Es importante mencionar que utilizaremos una variable que analice si dos lámparas consecutivas son o no disjuntas, pues, para que el intervalo $[a, b]$ sea cubierto completamente, requerimos que dicho par de lámparas compartan un espacio, así sea el punto de fin de una y el inicio de la otra, pues, por pequeña que sea, la solución no será admisible si existe alguna superficie sin cubrir.

```
1  int solucion()
2  {
3      int n, i, j, cantidad = 1;
4      pair <int, int> ultima_lampara;
5      vector<pair<int, int>> lamparas;
6      bool disjuntos;
7
```

```
8       cin>>n;
9       lamparas.resize(n);
10      for (i = 0; i < n; i = i + 1)
11          cin>>lamparas[i].first>>lamparas[i].second;
12      sort(lamparas.begin(), lamparas.end());
13
14      i = 0;
15      while (i < lamparas.size())
16      {
17          ultima_lampara = lamparas[i];
18          disjuntos = true;
19          for (j = i + 1; j < lamparas.size(); j = j + 1)
20          {
21              //No es posible cubrir todo el túnel.
22              if (lamparas[j].first > ultima_lampara.second)
23                  break;
24
25              //Buscamos la lámpara con mayor alcance en ese punto.
26              if (lamparas[j].second > lamparas[i].second)
27              {
28                  i = j;
29                  disjuntos = false;
30              }
31          }
32
33          if (disjuntos)
34              break;
35
36          cantidad = cantidad + 1;
37      }
38
39      if (disjuntos)
40          return -1;
41      return cantidad;
42  }
```

4.4. Programación Dinámica

¿Cuántos asteriscos ves en la siguiente línea?

* * * * * *

¿Y si aumentamos uno?

* * * * * * *

Gracias a la respuesta de la primera línea de asteriscos, no es necesario contar los de la segunda, pues si ya sabemos que en la primera hay 6, es fácil deducir que en la segunda hay $6 + 1 = 7$. Este ejemplo tan sencillo nos muestra el principio básico de la programación dinámica: memorizar la respuesta a cada paso y utilizarla para obtener la solución de una versión más grande del problema.

La programación dinámica (o DP, por sus siglas en inglés) resuelve problemas al recurrir a soluciones de subproblemas (algo similar a *Greedy* y a *Divide y Vencerás*); solemos utilizar este método cuando el problema a resolver puede ser dividido en subproblemas de manera sucesiva, mismos que heredan las características del problema original, y en este caso nos importará todo lo que ocurra en el pasado para decidir cuáles son las mejores decisiones presente y en el futuro.

Pensemos, por ejemplo, en la clásica función de Fibonacci. Ésta se puede obtener de manera recursiva, en donde, para obtener F_n, utilizamos la expresión $F_{n-1} + F_{n-2}$; la versión sencilla (sin programación dinámica) del algoritmo de la función de Fibonacci se muestra a continuación.

```
1  long long Fibonacci(long long n)
2  {
3      if (n <= 1)
4          return n;
5
6      return Fibonacci(n - 1) + Fibonacci(n - 2);
7  }
```

El problema con el programa anterior es que siempre tendremos que recalcular todos los F_i necesarios. Por ejemplo, si se realizan tres consultas: una para obtener F_6, otra para obtener F_7 y otra para obtener F_8, la función se ejecutará una vez para cada caso, y ya que cada caso hace una llamada a la misma función de los dos elementos anteriores, para k consultas, la complejidad de esta solución sería $O(k \cdot 2^n)$ (la

complejidad de la función de Fibonacci es $O(2^n)$, para el n-ésimo elemento, porque a cada paso hace una llamada a la función correspondiente a $i-1$ y a $i-2$). Cuando introducimos programación dinámica, introducimos la ventaja de memorizar las soluciones más pequeñas para utilizarlas en aquéllas más grandes. En nuestro ejemplo, al obtener F_6, habremos pasado por F_5, y si guardamos ambos, podremos obtener directamente $F_7 = F_6 + F_5$. Nuevamente, almacenando F_7, es fácil conocer el valor de $F_8 = F_7 + F_6$. Con estos cambios, la complejidad de la función de Fibonacci se reduce a $O(n)$, obteniendo hasta el n-ésimo término, y para k consultas, tendremos una complejidad de $O(k \cdot n)$. El fragmento de programa que se muestra abajo nos enseña a introducir la propiedad dinámica a la función de Fibonacci.

```
1  //Para almacenar todos los F_i, hasta 90.
2  //Debe llenarse de -1.
3  long long TablaFibonacci[90];
4
5  long long DP_Fibonacci(long long n)
6  {
7      //Si ya se calculó antes.
8      if (TablaFibonacci[n] != -1)
9          return TablaFibonacci[n];
10
11     //Casos base: 0, 1.
12     if (n <= 1)
13     {
14         TablaFibonacci[n] = n;
15         return TablaFibonacci[n];
16     }
17
18     //Si ninguna condición anterior se cumple:
19     //Llenamos F_n con la suma de los dos elementos anteriores.
20     TablaFibonacci[n] = DP_Fibonacci(n - 1) + DP_Fibonacci(n - 2);
21     return TablaFibonacci[n];
22 }
```

La función recursiva es sencilla de programar, pero debido al hecho de que no lleva un registro de los cálculos realizados previamente, su número de operaciones resulta en 2^n; la programación dinámica, sin embargo, cuenta con una desventaja frente al método convencional que no incluye memorización, y es que requiere mayor memoria, por lo que debes asegurarte de contar con suficiente espacio disponible.

Otro ejemplo notable en la aplicación de la programación dinámica está dentro de los coeficientes binomiales. En el capítulo de *Análisis Combinatorio* vimos las limitaciones del cálculo de coeficientes binomiales con grandes valores. Afortunadamente, en el mismo capítulo estudiamos una forma recursiva de calcular un coeficiente binomial, que surge de la comprensión del triángulo de Pascal:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } n > k \text{ y } k \neq 0 \\ \\ 1 & \text{si } k = 0 \text{ o } k = n \end{cases}$$

Ya que contamos con una expresión recursiva, podemos aplicar programación dinámica para almacenar los coeficientes anteriores y, en base a ellos, obtener coeficientes más grandes, esto sin la necesidad de operar con factoriales; gráficamente, podemos ver la tabla de almacenamiento de los coeficientes binomiales como se muestra a continuación.

	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$\cdots$
$n=0$	$\binom{0}{0}$					
$n=1$	$\binom{1}{0}$	$\binom{1}{1}$				
$n=2$	$\binom{2}{0}$	$\binom{2}{1}$	$\binom{2}{2}$			
$n=3$	$\binom{3}{0}$	$\binom{3}{1}$	$\binom{3}{2}$	$\binom{3}{3}$		
$n=4$	$\binom{4}{0}$	$\binom{4}{1}$	$\binom{4}{2}$	$\binom{4}{3}$	$\binom{4}{4}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$

```
1  //Para guardar los coeficientes utilizaremos una matriz.
2  //Hay que llenarla de -1.
3  long long TablaBinomial[60][60];
4
5  long long DP_Binomial(int n, int k)
6  {
7      //Coeficientes ya obtenidos.
```

```
8       if (TablaBinomial[n][k] != -1)
9           return TablaBinomial[n][k];
10
11      //Casos base.
12      if (n == k or k == 0)
13      {
14          TablaBinomial[n][k] = 1;
15          return TablaBinomial[n][k];
16      }
17
18      //El resto de los casos.
19      TablaBinomial[n][k] = DP_Binomial(n - 1, k - 1) + DP_Binomial(n - 1, k);
20      return TablaBinomial[n][k];
21  }
```

El análisis es similar al que hicimos con la función de Fibonacci, sólo que en esta ocasión haremos $n \cdot k$ cálculos, que es, de hecho, el fragmento de la tabla requerido para calcular el máximo valor que necesitemos.

4.4.1. Método Iterativo

Otra forma de incluir programación dinámica en un algoritmo es a través de métodos iterativos, en los que, en lugar de hacer uso de una función recursiva, llenaremos una tabla directamente con todas las respuestas, sin importar que éstas se usen o no.

Antes de hacer uso de esta técnica, es necesario tener en cuenta dos cosas: se recorrerá todo el espacio de búsqueda aunque no sea necesario, y requerimos conocer el comportamiento de las transiciones. Considera siempre los siguientes puntos.

- Determinar los parámetros que definen los estados del problema.
- Preparar una tabla de memorización de dimensiones iguales a los parámetros de cada estado, e inicializar únicamente los casos base.
- A partir de los casos base, determinar qué estados pueden ser llenados después, y repetir este proceso hasta que la tabla se haya llenado completamente. (Suelen

usarse tantos ciclos de repetición anidados como dimensiones tiene la tabla de memorización).

El algoritmo de Fibonacci con el método iterativo de programación dinámica se muestra a continuación. Aquí se llena la tabla directamente, sin importar si en algún momento requeriremos alguno de los datos F_i, y únicamente haremos la consulta de la información que necesitemos; hay que tener en cuenta que el máximo número de Fibonacci que soportará un dato *long long* es F_{93}, si requerimos trabajar con números mayores, habremos de hacer uso de módulos.

```
long long TablaFibonacci[90];

void Llenar()
{
    int i;

    //F_0 = 0
    TablaFibonacci[0] = 0;

    //F_1 = 1
    TablaFibonacci[1] = 1;

    for (i = 2; i < 90; i = i + 1)
        TablaFibonacci[i] = TablaFibonacci[i - 1] + TablaFibonacci[i - 2];
}
```

Mientras que la obtención de los coeficientes binomiales obedece al siguiente algoritmo, en donde los casos básicos son $\binom{i}{0} = \binom{i}{i} = 1$; para el resto de los coeficientes utilizamos la fórmula del triángulo de Pascal.

```
long long TablaBinomial[60][60];

void Llenar()
{
    int i, j;

    for (i = 0; i < 60; i = i + 1)
        TablaBinomial[i][0] = TablaBinomial[i][i] = 1;
```

```
9
10      for (i = 2; i < 60; i = i + 1)
11          for (j = 1; j < i; j = j + 1)
12              TablaBinomial[i][j] = TablaBinomial[i - 1][j - 1]
13                                  + TablaBinomial[i - 1][j];
14  }
```

Ventajas y Desventajas: Recursivo vs Iterativo

¿Cuándo es mejor un método que otro? Eso, en realidad, depende de cuáles sean los requerimientos de la solución, así como de las condiciones en las que se desarrolla el problema. A continuación se muestran algunas de las características de ambos métodos, para que tú elijas el más conveniente en cada ocasión.

- ***Recursivo***
  - **Ventaja:** Es una manera natural de cambiar de búsqueda completa a programación dinámica, y computa subproblemas sólo cuando es necesario.
  - **Desventaja:** Es más lento debido al tiempo que requiere invocar una función recursiva, y requiere más memoria que el método iterativo.

- ***Iterativo***
  - **Ventaja:** Es mas rápido, ya que no se requiere revisitar subproblemas y cada consulta se realiza en tiempo constante; puede ahorrar memoria.
  - **Desventaja:** No es intuitivo y recorre todo el espacio de búsqueda aunque no sea necesario.

Aprender programación dinámica, pese a la teoría necesaria, requiere bastas horas de práctica, que te mostrarán cómo relacionar los conceptos de cierto problema dado

con el funcionamiento de la programación dinámica. En lo sucesivo, se presentarán algunos problemas que ejemplificarán dicha relación.

Ejemplo 144. Construye un algoritmo que devuelva el factorial, módulo 9876543210, de k consultas.

Solución. La obtención del factorial de cierto natural n tiene una complejidad lineal, y si trabajamos cada consulta como un dato independiente, nuestro algoritmo terminaría en $O(k \cdot n)$, no así con la programación dinámica.

Gracias a las propiedades que vimos en la sección de *Aritmética Modular* en el capítulo de *Teoría de Números*, sabemos que, para aplicar correctamente los módulos a nuestra solución, será necesario aplicar módulo a cada operación realizada.

El siguiente fragmento de programa es la solución recursiva del problema.

```
1  //Llenar de -1.
2  long long TablaFactorial[100];
3
4  long long DP_Factorial_Modulo(int n)
5  {
6      int mod = 9876543210;
7
8      if (TablaFactorial[n] != -1)
9          return TablaFactorial[n];
10
11     if (n == 0)
12     {
13         TablaFactorial[n] = 1;
14         return TablaFactorial[n];
15     }
16
17     TablaFactorial[n] = (n % mod) * DP_Factorial_Modulo(n - 1) % mod;
18     return TablaFactorial[n];
19 }
20
21 void main()
22 {
23     int k, n, i;
```

```
    vector<int> consultas;

    cin>>k;
    for (i = 0; i < k; i = i + 1)
    {
        cin>>n;
        consultas.push_back(n);
    }

    for (i = 0; i < k; i = i + 1)
        cout<<DP_Factorial_Modulo(consultas[i])<<"\n";
}
```

Y abajo se presenta la solución con el método iterativo.

```
long long TablaFactorial[100];

long long Llenar(int n)
{
    int i, mod = 9876543210;

    TablaFactorial[0] = 1;

    for (i = 1; i <= n; i = i + 1)
        TablaFactorial[i] = (i % mod) * TablaFactorial[i - 1] % mod;
}

void main()
{
    int k, n, i;
    vector<int> consultas;

    for (i = 0; i < k; i = i + 1)
    {
        cin>>n;
        consultas.push_back(n);
    }

    for (i = 0; i < k; i = i + 1)
        cout<<Llenar(consultas[i])<<"\n";
}
```

A este punto, ya es claro que no es posible trabajar grandes coeficientes binomiales sin hablar de módulos y que puede ser muy conveniente involucrar programación dinámica para su cálculo. Con el fin de poner en práctica lo anterior, recordaremos un problema que ya hemos visto en el capítulo de *Análisis Combinatorio*.

Ejemplo 145. Resuelve el ejemplo 14 del capítulo de *Análisis Combinatorio*, considerando que C puede crearse por hasta 10^3 caracteres distintos, dado cierto número natural k y el tamaño de C: t. Ya que la respuesta puede ser muy grande, encuentra su equivalencia módulo 123456789.

Solución. Cuando trabajamos este ejemplo, encontramos que la solución responde a la expresión:

$$\binom{t}{k} k!$$

Misma que logramos reducir. En este caso, no utilizaremos esa simplificación, pues al buscar mostrar el uso de la programación dinámica, nos resulta más conveniente utilizar el comportamiento de los coeficientes binomiales y los factoriales en lugar de la expresión final propuesta en el ejemplo 14.

```
long long TablaBinomial[1000][1000];
long long TablaFactorial[1000];
int mod = 123456789;

long long DP_Binomial_Modulo(int n, int k)
{
    if (TablaBinomial[n][k] != -1)
        return TablaBinomial[n][k];

    if (n == k or k == 0)
    {
        TablaBinomial[n][k] = 1;
        return TablaBinomial[n][k];
    }

    TablaBinomial[n][k] = DP_Binomial_Modulo(n - 1, k - 1) +
                          DP_Binomial_Modulo(n - 1, k) % mod;
    return TablaBinomial[n][k];
}

```

```
21  long long DP_Factorial_Modulo(int n)
22  {
23      if (TablaFactorial[n] != -1)
24          return TablaFactorial[n];
25
26      if (n == 0)
27      {
28          TablaFactorial[n] = 1;
29          return TablaFactorial[n];
30      }
31
32      TablaFactorial[n] = (n % mod) * DP_Factorial_Modulo(n - 1) % mod;
33      return TablaFactorial[n];
34  }
35
36  long long main()
37  {
38      int k, t;
39      cin>>k>>t;
40
41      return DP_Binomial_Modulo(t, k) * DP_Factorial_Modulo(k) % mod;
42  }
```

Ejemplo 146. Resuelve el problema 43 (ubicado en el capítulo de *Análisis Combinatorio*) con programación dinámica. Devuelve la respuesta módulo 10007.

Solución. La solución que se desarrolla en el ejemplo 43 no nos proporciona una solución recursiva, sin embargo, el teorema 7 del capítulo de *Análisis Combinatorio* nos brinda una solución equivalente a través de una expresión de este tipo, misma que se muestra a continuación.

$$D_n = \begin{cases} 0 & \text{si } n = 1 \\ 1 & \text{si } n = 2 \\ (n-1)(D_{n-2} + D_{n-1}) & \text{si } n \geq 3 \end{cases}$$

```
1  int T_Derangements[100000];
2  int mod = 10007;
3
4  int solucion(int n)
5  {
6      if (T_Derangements[n] != -1)
```

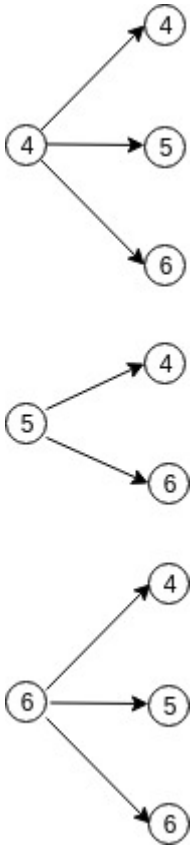
```
7           return T_Derangements[n];
8
9       if (n == 1)
10      {
11          T_Derangements[n] = 0;
12          return T_Derangements[n];
13      }
14
15      if (n == 2)
16      {
17          T_Derangements[n] = 1;
18          return T_Derangements[n];
19      }
20
21      T_Derangements[n] = ( ( (n - 1) % mod ) * ( solucion(n - 2)
22                          + solucion(n - 1) ) % mod ) % mod;
23      return T_Derangements[n];
24  }
```

Ejemplo 147. ¿Cuántos enteros positivos de exactamente n dígitos existen que estén formados por únicamente números 4, 5 y 6, tales que no contengan dos o más 5's contiguos? Devuelve la respuesta módulo 100019.

Solución. Pensemos primero en algunos ejemplos pequeños que nos permitan buscar un patrón. Sea $f(n)$ la cantidad de números naturales de n dígitos que cumplen con las características requeridas.

Cuando $n = 1$, $f(n) = f(1) = 3$ (4, 5 y 6). Cuando $n = 2$, $f(n) = 8$ (44, 45, 46, 54, 56, 64, 65 y 66). Nota que es posible construir los números de dos dígitos a partir de los de un dígito: En el caso de 4 y 6, podemos añadirle cualquiera de los tres dígitos (4, 5, 6), sin embargo, a 5 únicamente le podemos agregar 4 o 6, por tanto, es posible escribir a $f(2)$ como $f(2) = 3(2) + 2(1)$, en donde 2 son los números a los que se les permite agregar 4, 5 y 6, y 1 los que sólo permiten 4 y 6.

Para facilitar nuestro trabajo, introduciremos a la función $p(n)$, que representará la cantidad de números con n dígitos a los que *no* se les puede añadir un 5, es decir, aquéllos números que terminan justamente en 5. Así pues, podemos escribir a $f(2)$ como $f(2) = 3 \cdot [f(1) - p(1)] + 2 \cdot p(1)$.



Para $n = 3$, partiremos de lo obtenido en $f(2) = 8$ y $p(2) = 2$ (como se muestra en la imagen de arriba). Todos los números contemplados en $p(2)$ podrán ser concatenados únicamente con dos de las tres opciones, mientras que el resto, $f(2) - p(2)$, tienen todas las opciones, por tanto, $f(3) = 3 \cdot [f(2) - p(2)] + 2 \cdot p(2)$.

En general, la cantidad de números con n dígitos que se pueden formar con los requerimientos pedidos es:

$$f(n) = 2 \cdot p(n-1) + 3 \cdot [f(n-1) - p(n-1)]$$

La interrogante que debemos responde ahora es cómo obtener $p(n)$. Separemos los números que comprende $f(n)$ en dos grupos de acuerdo con su penúltimo dígito, es decir, de acuerdo a su terminación al obtener

$f(n-1)$: Los que terminan en 5 (y pertenecen a $p(n-1)$) y el resto. Todos los números que pertenezcan a $p(n-1)$ se bifurcarán en dos cadenas distintas (una a la que se le agrega 4 y otra a la que se le agrega 6), así que estos no pueden pertenecer a $p(n)$ y hay que eliminarlas restando el doble de $p(n-1)$. Luego, aún tenemos $f(n) - 2 \cdot p(n-1)$ números por analizar, pero sabemos ninguno de ellos tiene como penúltimo dígito a 5, lo que implica que el último puede ser cualquiera entre 4, 5 y 6; así, una tercera parte de $f(n) - 2 \cdot p(n-1)$ terminará en 4, una tercera parte en 5 y una tercera parte en 6, y ya que únicamente requerimos los números terminados en 5:

$$p(n) = \frac{f(n) - 2 \cdot p(n-1)}{3}$$

Con esto, concluimos que la solución de nuestro problema recae en las siguientes dos expresiones.

$$f(n) = \begin{cases} 3 & \text{si } n = 1 \\ 2 \cdot p(n-1) + 3 \cdot [f(n-1) - p(n-1)] & \text{para el resto de los casos} \end{cases}$$

$$p(n) = \begin{cases} 1 & \text{si } n = 1 \\ \frac{1}{3}\Big[f(n) - 2 \cdot p(n-1)\Big] & \text{para el resto de los casos} \end{cases}$$

```
1  pair<int, int> Tablas[1000000];
2  //El primer elemento será f y el segundo p.
3
4  int mod = 100019;
5
6  void Llenar_Tablas()
7  {
8      int i;
9
10     Tablas[1].first = 3;
11     Tablas[1].second = 1;
12
13     for (i = 2; i < 1000000; i = i + 1)
14     {
15         Tablas[i].first = ( (2 * Tablas[i-1].second % mod)
```

```
16                          + 3 * ( Tablas[i-1].first - Tablas[i-1].second
17                          + mod ) % mod ) % mod;
18
19          Tablas[i].second = ( (Tablas[i].first - 2 * Tablas[i-1].second)
20                           % mod + mod) % mod ) * ExpRapida_modulo(3,
21                           mod-2, mod) % mod;
22      }
23  }
24
25  void main()
26  {
27      int n;
28      Llenar_Tablas();
29
30      cin>>n;
31      cout<<Tablas[n].first<<"\n";
32  }
```

En este caso, tuvimos que hacer uso de uno de los métodos que involucran divisiones en aritmética modular (vistos en *Teoría de Números*), invocando con ello el tema de *exponenciación rápida*.

Si tienes alguna duda sobre la aplicación de los módulos en un programa, acude a la sección de *Aritmética Modular* de esta obra.

Ejemplo 148. Dada una serie de números naturales tales que cada uno de ellos es el producto de factores primos menores que 30, encuentra la subsecuencia más grande cuyo producto sea un cuadrado perfecto, considerando que dicha serie puede tener hasta 10^6 elementos. (Problema tomado de [24]).

Solución. Un número es un cuadrado perfecto si y sólo si en su descomposición canónica únicamente hay exponentes pares presentes, sin importar cuáles sean.

Los factores primos divisores de números de la serie dada pertenecen al conjunto $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}$, y todo número perteneciente a la lista se puede escribir de la forma

$$2^{a_2} \cdot 3^{a_3} \cdot 5^{a_5} \cdot 7^{a_7} \cdot 11^{a_{11}} \cdot 13^{a_{13}} \cdot 17^{a_{17}} \cdot 19^{a_{19}} \cdot 23^{a_{23}} \cdot 29^{a_{29}}$$

Es claro que los exponentes pueden tomar cualquier valor entero, sin embargo, lo que realmente nos importa de ellos es la paridad (pues buscamos un producto cuya descomposición canónica tenga sólo exponentes pares), por lo que a cada a_i le aplicaremos módulo 2 con el fin de quedarnos únicamente con 0's y 1's. Si tuviéramos, por ejemplo, los números 8, 9, 15, y 135, los exponentes de su descomposición canónica se verían como sigue.

	29	23	19	17	13	11	7	5	3	2
8	0	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	1	1	0
135	0	0	0	0	0	0	0	1	1	0

$8 = 2^3$, y ya que el exponente es impar, es congruente con 1, módulo 2, y el resto de sus exponentes se mantiene en 0; $9 = 3^2$, por lo que todos sus exponentes son congruentes con 0, módulo 2; $15 = 3 \cdot 5$ y $135 = 3^3 \cdot 5$, con lo que, en nuestra representación de exponentes, son iguales.

El anterior ejemplo es una clara prueba de que con esta representación puede suceder que varios de los números dados sean iguales (con lo que los 10^6 números se pueden reducir a 2^{10} números distintos en base 2), de lo que podemos sacar ventaja para encontrar productos de exponentes pares, pues al multiplicar los números originales, estaremos sumando los exponentes, lo que en este caso, realizaremos con una operación XOR ($\wedge$), ya que al sumar cualesquiera dos elementos iguales, resultará un exponente par (es decir, 0), contrario a cuando sumamos dos elementos distintos (1).

Ahora bien, parecería buena idea "agrupar" tantos números binarios iguales como sea posible; si la cantidad es impar, es obvio que sobrará uno, pero si es par, puede pasar (o no) que la mejor elección no sea tomar todos los posibles. Por ejemplo, pensemos en una serie de enteros cuya representación es 3 (8 veces), y $3 \cdot 7$, $7 \cdot 11$ y 11 (una vez cada uno). Si agrupamos los 3's, tendremos 8 números, y dado que el producto del res-

to no puede formar un cuadrado perfecto, nuestra subsecuencia tendrá una cardinalidad de esos ocho elementos. No obstante, si en lugar de ello tomamos seis 3's, podremos multiplicar uno de los restantes por $3 \cdot 7$, el resultado con $7 \cdot 11$, y esto a su vez, por 11, así, nuestra subsecuencia estaría compuesta por diez enteros.

Sea $f(i, num)$ el tamaño de la mayor subsecuencia (creada por las representaciones que hemos estipulado arriba) que inicia en el índice i y va al final de la serie, y que, al sumarse con la representación de los exponentes de num, forma un cuadrado perfecto.

Luego, si la cantidad de representaciones i es par, la función $f(i, num)$ surgirá de elegir la máxima de las subsecuencias de entre las siguientes dos opciones:

- $f(i+1, num) + cantidad[i]$, el tamaño de la mayor subsecuencia que comienza en $i+1$ más todos los números binarios de la forma i.
- $f(i+1, num \wedge i) + cantidad[i] - 1$, el tamaño de la mayor subsecuencia que comienza en $i+1$, sumado a la cantidad menos uno de i, pues, como en el ejemplo de arriba, requeriremos agrupar uno de los $i's$ restantes con num para tomar más elementos de la serie.

Por otro lado, si $cantidad[i]$ es impar, nuestras opciones serán:

- $f(i+1, num) + cantidad[i] - 1$, que implica que tomaremos todos los números de la forma i excepto uno, y los sumaremos a la subsecuencia $f(i+1, num)$.
- $f(i+1, num \wedge i) + cantidad[i]$, que implica agrupar tantos números i como se puede, y el restante sumarlo con num.

```
1  #define tam (1<<10)
2  int tabla[tam][tam], cantidad[tam];
3
4  int f(int i, int num)
5  {
6      if (i == tam)
```

```
7       {
8           if (num == 0)
9               return 0;
10          return -INF;
11      }
12
13      int &respuesta = tabla[i][num];
14      if (~respuesta)
15          return respuesta;
16
17      if (cantidad[i])
18      {
19          if (cantidad[i] % 2 == 0)
20              respuesta = max( f(i + 1, num) + cantidad[i],
21                      f(i + 1, num ^ i) + cantidad[i] - 1);
22          else
23              respuesta = max( f(i + 1, num) + cantidad[i] - 1,
24                      f(i + 1, num ^ i) + cantidad[i]);
25
26          return respuesta;
27      }
28
29      return respuesta = f(i + 1, num);
30  }
```

Conclusiones

Aunque existen numerosos libros y textos para aprender a programar, y que abordan varios elementos matemáticos requeridos en el área, el principal objetivo de esta obra es mostrar al lector el importante papel que juegan las matemáticas al elaborar algoritmos eficientes, así como conducirlo desde conceptos básicos (a un nivel licenciatura) en combinatoria, teoría de números y teoría de grafos, y llevarlo paulatinamente a resultados más sofisticados, de los que conocerá las bases matemáticas y cuya aplicación podrá llevar a la práctica dentro de una solución algorítmica.

A la vista de los textos disponibles para el área, surge la idea de construir esta obra, un material que tuviera la capacidad de cubrir los conocimientos matemáticos básicos e intermedios que un participante de programación competitiva requiere, con lo que se busca facilitar el aprendizaje autodidacta de los estudiantes, quienes, en este trabajo, tendrán acceso no sólo a los algoritmos clásicos, sino también a las bases matemáticas que los sustentan, así como a ejemplos que tornen palpable su aplicación.

Como se dijo antes, en este material se presentan vastos resultados que, sin duda, se encontrarán en muchos otros libros, sin embargo, aquí se teorizan también conceptos que no fueron hallados en otras referencias, se muestran problemas y soluciones propios de la autora y se proponen algoritmos alternativos a los antes publicados. Con ello, de manera paralela, se ha colaborado con la elaboración de problemas al Juez GUAPA [3], creado en la FES Acatlán, y los concursos internos realizados en la institución, así como a la Región de México y Centroamérica del ICPC en las fases clasificatorias (algunos de los cuales se presentan en la sección de *Anexos* de esta

obra). Además, se ha hecho una colaboración inicial con la Escuela Nacional Preparatoria 9 de la UNAM, a la que se le ha proporcionado material de entrenamiento similar a los ejemplos desarrollados a lo largo de esta obra.

Este trabajo es el resultado de varios años de entrenamiento y participación en concursos de programación competitiva y de matemáticas de la autora, experiencia que también fue utilizada en la instrucción de miembros del Grupo Universitario de Algoritmia y Programación Competitiva (GUAPA) de la FES Acatlán entre 2016 y 2018, quienes, a este punto, ya han asegurado su participación en la Final Regional de México del ICPC 2019; cabe mencionar que, para clasificar, los equipos tuvieron que resolver problemas relativos a combinatoria y teoría de números, materias en las que fueron entrenados por la autora de este material.

Si bien esta obra fue escrita con la intención de ayudar a la formación y el entrenamiento de los alumnos de la FES Acatlán involucrados en el mundo de la programación competitiva, se espera que este material sea de utilidad a todas las instituciones participantes en este tipo de eventos, y que sirva como apoyo para cualquier estudiante de una licenciatura afín a Matemáticas y Computación.

Escribir este trabajo ha requerido de la superación de varias fases. La primera, indiscutiblemente, fue el proceso de aprendizaje de las áreas abordadas aquí, que dicho sea de paso, fue de varios años; el segundo reto fue encontrar la manera de transmitir el conocimiento a un grupo de estudiantes, de tal forma que realmente entiendan y sean capaces de aplicar (y a su tiempo, transmitir) la teoría matemática a la hora de diseñar algoritmos eficientes, y este punto fue particularmente complicado, pues parte del reto era también comprender que todos ellos aprenderían y asimilarían el conocimiento de distinta manera, y que no todos serían atraídos de la misma forma hacia el área matemática y hacia el área de programación.

El Grupo Universitario de Algoritmia y Programación Competitiva, en este punto, es conformado únicamente por estudiantes de licenciatura, y son ellos mismos quienes se encargan de la instrucción de otros miembros, por lo que una de las complicaciones internas que se presentan es que no existe un temario a seguir, y es justo uno de los desafíos a los que la autora se enfrentó y con los que este trabajo pretende ayudar.

De la misma forma en que fue complicado comprender varios de los conceptos vistos

aquí, fue complicado escribir definiciones, proposiciones, demostraciones y ejemplos que fueran lo más fácilmente comprensibles (dentro de lo posible); en específico, para escribir problemas, tanto para esta obra como para diversos concursos, se requiere la creación de una historia, de asegurar una solución (al menos en C) y de prever la dificultad, el tamaño de los datos y los segundos que serán permisibles, lo que requiere de bastante dedicación y tiempo.

Proponer problemas para la Región de México y Centroamérica fue uno de las fases más retadoras y satisfactorias, pues, si bien el trabajo que implicó fue grande, también probó que el conocimiento y los problemas que se proponen aquí tienen el nivel necesario para figurar dentro de los concursos de programación más importantes del país.

El mayor reto de todos fue aceptar que no sería posible recabar todo el conocimiento necesario y delimitar los alcances de la obra, sin embargo, este hecho también da una pauta para pensar en el trabajo a futuro; en específico, se piensa en incluir un capítulo de Geometría y ampliar los temas vistos en cada capítulo, así como los ejemplos de cada área. A su vez, se busca su difusión dentro de todas las instituciones del país relacionadas con el área de matemáticas y de programación, y si es posible, también la distribución internacional.

Anexos

Análisis de Algoritmos

El funcionamiento de un código se mide de dos maneras: por el tiempo que tarda en ejecutarse y la memoria que requiere. El lenguaje utilizado para esta medición se conoce como *Big O*, y se utiliza la simbología de $O(k)$, en donde k es el número de operaciones requeridas para la ejecución del programa (si hablamos de tiempo) o la memoria utilizada (si hablamos de memoria). Ya que en los concursos de programación competitiva uno de los aspectos decisivos es la eficiencia de las soluciones propuestas, enfocaremos nuestra atención al factor tiempo, esto siempre bajo la premisa de que una computadora promedio puede realizar únicamente 10^8 operaciones por segundo [2]. En [25] encontrarás la cantidad de segundos que suele darse a cada lenguaje, aunque esto puede variar dependiendo del juez en que estés trabajando.

Hablar de análisis de algoritmos implica evaluar la rapidez con la que se desarrolla nuestra solución, y estas consideraciones pueden ser la clave para la aceptación de un problema, e incluso para la obtención de un buen lugar, sin contar claro, con el hecho de que este conocimiento será útil, ya sea en un ámbito académico o en un ámbito laboral. Aquí hablaremos únicamente de algunos ejemplos sencillos y veremos la teoría para la comprensión básica del análisis de algoritmos, que podrás ampliar si consultas [26]; si te interesa profundizar aún más, acude a [27].

Comencemos pensando en un ejemplo sencillo: ¿Cuántas operaciones se requieren para la lectura de n datos? Independientemente de qué ciclo de repetición uses para

recibirlos, éste será un ciclo que iterará n veces, por lo que podemos decir que la complejidad de este procedimiento es $O(n)$:

```
for (i = 0; i < n; i = i + 1)
    cin>>dato[i];
```

Si en lugar de eso, buscamos llenar una matriz de $n \times n$, entonces tendremos que utilizar dos ciclos, uno anidado dentro del otro: uno para recorrer las columnas y el otro para recorrer las filas, lo que implica una complejidad de $O(n^2)$, pues para cada valor de i, habrá que hacer n operaciones (iteradas en j):

```
for (i = 0; i < n; i = i + 1)
    for (j = 0; j < n; j = j + 1)
        cin>>dato[i][j];
```

Volviendo al ejemplo en el que leemos n datos, si, además de leerlos (y almacenarlos), agregamos la característica de que sean enteros y deseamos obtener la suma de todos ellos, tendremos dos opciones: una en la que agregamos una sentencia al ciclo de repetición en que se reciben los datos, o otra en la que, en ciclos distintos, se reciben los datos y se realiza la suma:

```
suma = 0;
for (i = 0; i < n; i = i + 1)
{
    cin>>dato[i];
    suma = suma + dato[i];
}
```

```
for (i = 0; i < n; i = i + 1)
    cin>>dato[i];

suma = 0;
for (i = 0; i < n; i = i + 1)
    suma = suma + dato[i];
```

¿Cuál de los dos códigos resulta más rápido? El primero de ellos tiene dos instrucciones dentro del ciclo de iteración, lo que significa que se harán $2n$ operaciones, y contando la inicialización de la variable *suma*, tendremos una complejidad de $O(2n+1)$. El segundo código, por otro lado, realiza n operaciones en el primer ciclo, y posteriormente, en el segundo ciclo, realiza otras n; así, y contando la inicialización de *suma*, la complejidad será de $O(n+n+1) = O(2n+1)$. Con esto podemos ver que la complejidad en tiempo de ambos ejemplos es equivalente.

Ahora bien, a pesar de que contamos $2n + 1$ operaciones en los códigos arriba, diremos que su complejidad es de $O(n)$, pues podemos suprimir todas las constantes. La razón es que la métrica de *Big O* describe cómo escala el tiempo de ejecución, y las constantes no tendrán un impacto significativo. Así, diferentes algoritmos con complejidades $O(3n + 100000)$, $O(500n)$, $O(2n + 16)$, serán catalogados como algoritmos de complejidad *lineal* con respecto a n, es decir, $O(n)$. Lo miso ocurrirá con aquellos códigos de complejidad cuadrática: $O(n^2 + 12)$, $O(2n^2)$, $O(14n^2 + 10^5)$, etcétera, tendrán complejidad $O(n^2)$. De la misma manera podemos continuar con los ejemplos, pero ahora sólo diremos que eliminaremos siempre las constantes.

Así como eliminamos las constantes, podemos eliminar los términos no dominantes. Por ejemplo, para un algoritmo cuya complejidad es $O(n^2 + n)$, diremos que simplemente tiene complejidad $O(n^2)$. Para entender por qué, piensa en un código que realiza $2n^2$ operaciones, es decir, n^2+n^2. Al eliminar la constante, "descartamos" un n^2, y si no nos importa un n^2, ¿por qué nos importaría cualquier término menor a él, en este caso, n?

Ten siempre en cuenta que es válido eliminar constantes y términos no dominantes, pero si obtienes una complejidad compuesta por más de una variable ambas deben estar presentes, salvo que tengas conocimiento de una relación entre ellas. Por ejemplo, para un código de complejidad $O(m + n)$, tenemos que conservar toda la expresión, sin embargo, si tenemos certeza de que m, a lo más, puede tomar el valor de n, podemos decir que $O(m + n) = O(2n) = O(n)$.

Pensemos ahora en que debemos obtener la suma de los primeros n números naturales. Si decides delegar todo el trabajo de la suma a la computadora, seguramente programarás algo muy similar al código mostrado a continuación.

```
1  suma = 0;
2  for (i = 0; i < n; i = i + 1)
3      suma = suma + i;
```

El problema con esta solución es que el tiempo se incrementará en tanto se incremente el valor de n, con lo que, para un entero mayor que 10^8 este código no cumplirá con el requerimiento del tiempo. Sin embargo, si tus conocimientos matemáticos abarcan la fórmula de Gauss, sabes que esa misma suma se resume a la fórmula $\frac{n(n+1)}{2}$, cuya

solución es *constante*, lo que expresaremos como $O(1)$, y que implica que el tiempo en que se ejecute no cambiará con el crecimiento de n. Éste sólo es uno de infinitos ejemplos que muestran cómo el trabajo matemático mejora de manera decisiva la eficiencia de una solución.

Hasta ahora, hemos visto pequeños ejemplos de códigos con complejidades constantes, lineales y cuadráticas, no obstante, existen muchas más. Una búsqueda binaria (que podrás encontrar en el capítulo de *Técnicas Avanzadas para la Resolución de Problemas*), por ejemplo, tiene una complejidad $O(log\ n)$; por otro lado, un algoritmo recursivo de los números de Fibonacci es de $O(2^n)$, pues, F_i se requiere la consulta de los dos números anteriores, así, para F_n, necesitaremos 2^n consultas. A lo largo de esta obra, encontrarás varios ejemplos de distintas complejidades que te mostrarán lo diversos que pueden ser los tiempos en que se ejecuta un algoritmo.

Funciones más Utilizadas

Función	Tarea que realiza	Página
`potencia(base, pot)`	Eleva el número *base* a la potencia entera *pot*.	7
`factorial(n)`	Obtiene $n!$.	10
`binomial(n, k)`	Obtiene el coeficiente binomial $\binom{n}{k}$.	15
`multinomial(n, V)`	Obtiene coeficiente multinomial $\binom{n}{k_1,k_2,\ldots,k_v}$, en donde cada k_i es un elemento de vector V.	26
`C(m, n)`	Obtiene recursivamente, con la fórmula de Pascal, $\binom{m}{n}$.	48
`primos_relativos(n)`	Cuenta los primos relativos de n menores que n.	67
`subfactorial(n)`	Devuelve el subfactorial correspondiente al natural n.	77
`D(n)`	Obtiene de manera recursiva el subfactorial de n.	80
`divisores(n)`	Devuelve todos los divisores de n.	94
`criba(n)`	Encuentra todos los primos menores o iguales que n.	99
`divsores_primos(n)`	Encuentra los divisores primos del entero n.	100
`desc_canonica(n)`	Obtiene la descomposición canónica de n.	101
`exponentes_primos(n)`	Obtiene los exponentes de la descomposición canónica de n.	104
`potencia_modulo(n, pot, mod)`	Realiza la operación n^{pot} % mod.	117

Función	Tarea que realiza	Página
factorial_modulo(mod)	Realiza el factorial hasta un cierto límite dado previamente aplicando módulo *mod* a cada paso. Se recomienda como un preproceso.	121
llenar_PReverso(mod)	Obtiene el producto $M(M-1)(M-2)\cdots(k+1)$, para cualquier k entre 1 y M, inclusive. A dicho producto se le aplica módulo *mod*. Se recomienda como un preproceso.	140
obtener_hash(cadena, t, B, mod)	Contiene la fórmula para obtener el hash de la primera subcadena de tamaño t de la cadena *cadena*. Dicho proceso se realiza en base B y módulo *mod*.	148
rolling_hash(cadena, hash, B, mod, t)	Obtiene el hash de las subcadenas de tamaño t de *cadena* utilizando la base B y el módulo *mod*. Ya que el procedimiento es recursivo, la función se alimenta del hash de la subcadena inmediatamente anterior.	148
A_Euclides(a, b)	Obtiene el máximo común divisor de a y b.	166
Euclides_Extendido(a, b)	Obtiene los enteros x, y para los que $mcd(a,b) = ax+by$.	172
obtener_mcm(a, b)	Regresa el mínimo común múltiplo de los enteros a y b.	179
exp_rapida(n, k)	Calcula n^k con el método de exponenciación rápida.	185
ExpRapida_modulo(n, k, mod)	Calcula $n^k \,\%\, mod$ con el método de exponenciación rápida.	186
Dijkstra_ruta(LA, s, t)	Aplica el algoritmo de Dijkstra al grafo cuya lista de adyacencia es LA, y encuentra la ruta de menor peso de s a t.	215
Dijkstra_peso(LA, S)	Aplica el algoritmo de Dijkstra al grafo cuya lista de adyacencia es LA, y encuentra el costo de la ruta de menor peso desde S a todos los vértices alcanzables.	217

Función	Tarea que realiza	Página
`FloydWarshall(M, P)`	Aplica el algoritmo de Floyd-Warshall al grafo cuya matriz de adyacencia es M, haciendo uso de la matriz de predecesores P.	230
`BellmanFord(LA, s)`	Aplica el algoritmo de Bellman-Ford al grafo cuya lista de adyacencia es LA, tomando como fuente al vértice s. La revisión de ciclos no negativos se realiza en la función *ciclo_negativo*, codificada en la misma página.	240
`Kruskal(LA)`	Devuelve el peso del árbol de expansión mínima del grafo cuya lista de aristas es LA, esto, aplicando el algoritmo de Kruskal.	251
`PrimJarnik(LA, s)`	Devuelve el peso del árbol de expansión mínima del grafo cuya lista de adyacencia es LA, esto, aplicando el algoritmo de Prim-Jarník y utilizando al vértice s como fuente.	258

Problemas usados en Concursos

En esta sección se presentan algunos de los problemas con los que se ha colaborado en diversos eventos, tanto dentro de FES Acatlán, como en concursos externos. Todos ellos requieren de cierto conocimiento y razonamiento matemático, y nos enfocaremos en esa parte de la solución, por lo que en esta ocasión, el código correrá por tu cuenta.

Dicho sea de paso, podrás encontrar todos los problemas aquí enlistados en [3].

Jaimina Party Invitations

Gran Premio de México 2019 - Primera fecha: Problema J

Jaimina está planeando una fiesta por su cumpleaños. Enviará invitaciones físicas a todos sus amigos, y para evitar desperdiciar, decidió hacer varias invitaciones de hojas de papel de $m \times n$ ($1 \leq m \neq n \leq 10^{16}$). Para ello, Jaimina hizo cortes en las hojas satisfaciendo tres condiciones:

1. Los cortes deben ser paralelos a los lados de las hojas.
2. Todos los rectángulos resultantes de una hoja deben tener las mismas dimensiones.
3. Jaimina dividirá todas las hojas por una cantidad de rectángulos que sea la potencia entera de un número primo p (todas las hojas se dividen en la misma cantidad de rectángulos): p^q ($2 \leq p \leq 10^{12}$ y $q \in \mathbb{N}$, $1 \leq q \leq 10^{18}$).

Una vez que tiene los rectángulos que usará para las invitaciones, escribe el mensaje y decora el perímetro de todos ellos con listones de colores.

Tu tarea es ayudarla a saber cuántos centímetros de listón debe comprar. Considera que Jaimina usará tantas hojas como configuraciones distintas sea posible crear, así que debes asegurarte de que los listones que compre serán suficientes para decorar todos los rectángulos de todos los posibles acomodos que cumplan con las condiciones mencionadas arriba. Cada configuración es obtenida exactamente una vez.

Devuelve la respuesta módulo $10^9 + 7$.

Solución. Para obtener r rectángulos de una hoja, todos de las mismas dimensiones, es necesario dividir el largo y el ancho de la hoja en a y b partes iguales, respectivamente, tales que $a \cdot b = r$. Así pues, al ser r una potencia de primos, podemos escribir a r como $r = p^k$ (para p primo y $k \in \mathbb{N}$), y las únicas posibles formas de crear los rectángulos satisfaciendo las condiciones requeridas son que el producto $a \times b$ adquiera los valores:

$$\begin{aligned} &p^0 \times p^k \\ &p^1 \times p^{k-1} \\ &p^2 \times p^{k-2} \\ &\vdots \\ &p^k \times p^0 \end{aligned}$$

El problema entonces se traduce en obtener la suma de los perímetros de todos los rectángulos que sea posible obtener con los productos enlistados arriba.

Primero, hay que tener claro cuántas configuraciones distintas es posible formar en las que se obtengan p^k rectángulos (esto es, cuántos productos hay en la lista de arriba): Si hay $k+1$ maneras de escribir a p^k como el producto de dos enteros positivos (y las hay, ya que los exponentes de p van de 0 a k), hay $k+1$ formas de dividir una hoja de papel en $r = p^k$ rectángulos.

Luego, hay que ver que, aunque todas esas configuraciones sean distintas, no es necesario recrearlas todas, pues sin importar cómo se divida la hoja, al sumar el perímetro de los rectángulos pequeños, lo que se hace en realidad es sumar m y n un número distinto de veces.

Sean p^i y p^{k-i} las divisiones que resultan de dividir la base y la altura de la hoja, respectivamente. Con ello, cada rectángulo obtenido será de dimensiones $\frac{m}{p^i} \times \frac{n}{p^{k-i}}$; sumar el perímetro de cada uno requiere considerar ambos lados dos veces. Dado que se trata de p^k rectángulos, la cantidad de listón que se requerirá en este caso es:

$$p^k\left[2\left(\frac{m}{p^i}+\frac{n}{p^{k-i}}\right)\right]=2\left[mp^{k-i}+np^i\right]$$

No obstante, recordemos que i tomará todos los valores entre 0 y k, inclusive, por lo que habrá que sumar el perímetro de los rectángulos que resulten de todas las configuraciones posibles, esto es:

$$\sum_{i=0}^{k}2(mp^{k-i}+np^i)=2\sum_{i=0}^{k}(mp^{k-i}+np^i)$$

Nota que $\sum_{i=0}^{k}p^i=\sum_{i=0}^{k}p^{k-i}$, por lo que:

$$2\sum_{i=0}^{k}(mp^{k-i}+np^i)=2\sum_{i=0}^{k}(mp^i+np^i)=2(m+n)\sum_{i=0}^{k}p^i$$

La suma $\sum_{i=0}^{k}p^i$ constituye una serie geométrica, por lo que la expresión obtenida arriba puede escribirse como sigue:

$$2(m+n)\left(\frac{p^{k+1}-1}{p-1}\right)$$

En este punto ya tenemos una fórmula funcional, no obstante hay que considerar dos aspectos importantes para la resolución completa de ésta, y están directamente relacionados con el tamaño de los datos de entrada.

En primer lugar, hay que aplicar módulos a cada procedimiento de la fórmula (el módulo no puede ser aplicado únicamente al final pues los datos se desbordarían), y habrá que hacer uso de un resultado del Pequeño Teorema de Fermat para resolver este problema (revisar *Método 1* de la subsección *Divisiones en Aritmética Modular* de esta obra) y, en lugar de multiplicar $2(m+n)(p^{k+1}-1)$ por $1/(p-1)$, realizaremos las operaciones:

$$2(m+n)(p^{k+1}-1)(p-1)^{10^9+5}$$

Aplicando los módulos necesarios.

En segundo lugar, se requiere un proceso de *exponenciación rápida* para resolver p^{k+1} y $(p-1)^{10^9+5}$. La función $ExpRap\_modulo(n,k,mod)$, descrita en la página 322, puede ayudarte con ello.

Credit Card PIN Number
Gran Premio de México 2019 - Segunda fecha: Problema C

Febo tiene una nueva tarjeta de crédito y debe crear un NIP para ella. Para crearlo, es necesario que se satisfagan dos condiciones:

1. El número debe tener exactamente K dígitos ($2 \leq K \leq 10^6$).
2. No debe haber dos (o más) dígitos iguales juntos.

Encuentra la cantidad de NIPs válidos de entre los que puede elegir Febo para su nueva tarjeta de crédito, y devuelve la respuesta módulo $10^9 + 7$.

Solución. Veamos a cada NIP como un arreglo de tamaño K. En la primera posición es posible colocar cualquiera de los 10 dígitos (0, 1, 2, ..., 9); en la segunda, para asegurar que el dígito que se escriba ahí no sea el mismo que el anterior, únicamente se tienen 9 opciones. Para la tercera posición, tenemos que asegurar que el dígito que coloquemos ahí no sea el mismo que el de la posición previa (sin importar el valor de la primera casilla), esto arroja 9 posibilidades.

Generalizando, para contar las posibilidades de cualquier casilla i, donde $2 \leq i \leq K$, únicamente debemos excluir el dígito que se colocó en la casilla previa, lo que implica reducir el número de dígitos posibles (10) en 1; no olvidemos que el caso en el que $i = 1$ es el único que contempla los 10 dígitos como válidos.

Así, por el *Principio Fundamental de Conteo*, el total del NIPs distintos que se puede crear es:

$$10 \times 9^{K-1}$$

No olvidemos que hay que aplicar módulos a cada operación, por lo que la función $ExpRap\_modulo(n, k, mod)$, descrita en la página 322, será de ayuda para la resolución de esa expresión.

Forgotten PIN Number
Gran Premio de México 2019 - Segunda fecha: Problema F

Febo es una persona tan distraída que olvidó el NIP de su nueva tarjeta de crédito a sólo unas horas de haberla recibido. Lo único que recuerda es que todos los dígitos en el NIP eran distintos.

Para crear un nuevo NIP, Febo debe completar algunos trámites en el banco: Un ejecutivo le proporciona el NIP olvidado, y por obvias razones de seguridad, éste debe ser cambiado. Como Febo se conoce y está consciente de su terrible memoria, decidió que el nuevo NIP se compondrá de los mismos dígtos que el anterior, con la condición extra de que dos caracteres que estaban juntos en el NIP original no deben estar juntos en el nuevo (en el mismo orden).

El banco es tan sofisticado que para hacer sus NIPs más seguros, usa un sistema numérico con 10^7 dígitos, sin embargo, no fuimos capaces de imprimirlos y, en lugar de ello, los identificaremos con números naturales entre 1 y 10^7.

Dado el NIP olvidado, ¿cuántas opciones (con los mismos dígitos que en el NIP olvidado) son excluidos para crear otro NIP? Encuentra la respuesta módulo $10^9 + 7$.

> **Solución.** Lo primero que se requiere es obtener el número de elementos de los que se compone el NIP. Para esto, habrá que leer cada NIP como una cadena y contar los espacios presentes en ellos. Ya que habrá un espacio menos que elementos, para obtener la cantidad de dígitos sólo obtendremos el número de espacios más 1. Llamaremos k al número de dígitos de un NIP. Una vez obtenido ese dato, procederemos por *Inclusión y Exclusión*.
>
> Nombremos F al NIP olvidado y N al nuevo. Sea P_i la propiedad de que, en el nuevo NIP, aparezcan juntos los elementos $F[i]$ y $F[i+1]$, para $1 \leq i \leq k-1$, y sea A_i el conjunto de NIPs que cumplen la condición P_i. Ya que se tienen k dígitos, podemos contar $k-1$ pares de caracteres contiguos, esto es, $k-1$ propiedades. Así, por el Principio de Inclusión y Exclusión, la cantidad de NIPs que cumplen con al menos una de las propiedades es:

$$\bigcup_{i=1}^{k-1} |A_i| = S_1 - S_2 + \cdots + (-1)^k S_{k-1}$$

En donde:

$$\begin{aligned}
S_1 &= |A_1| + |A_2| + \cdots + |A_{k-1}| \\
S_2 &= |A_1 \cap A_2| + |A_1 \cap A_3| + \cdots + |A_{k-2} + A_{k-1}| \\
S_3 &= |A_1 \cap A_2 \cap A_3| + |A_1 \cap A_2 \cap A_4| + \cdots + |A_{k-3} + A_{k-2} + A_{k-1}| \\
&\cdots \\
S_{k-1} &= |A_1 \cap A_2 \cap \cdots \cap A_{k-1}|
\end{aligned}$$

Para obtener el valor de los S_i, tomemos alguna de las propiedades, digamos P_i. Para contar las cadenas que se pueden formar cumpliendo (al menos) dicha condición, veremos a la pareja P_i como un solo elemento. De esta forma, tendremos $k-1$ dígitos para acomodar en $k-1$ lugares, en donde no se permiten repeticiones, lo que podemos contar como $(k-1)!$. Considerando que se tienen $k-1$ propiedades, distintas, $S_1 = (k-1)\cdot(k-1)!$.

Para obtener S_2, tomaremos las propiedades P_i y P_j $(i \neq j)$. Existirán dos posibilidades: Que dichas parejas sean disjuntas, o bien, que compartan exactamente un elemento. En el primer caso, tendremos $k-4+2 = k-2$ elementos para acomodar en $k-2$ lugares. En el segundo, habrá $k-3+1 = k-2$ dígitos para acomodar en $k-2$ posiciones. Dado que se pueden elegir $\binom{k-1}{2}$ pares de propiedades, $S_2 = \binom{k-1}{2} \cdot (k-2)!$.

En general, cuando se toman h propiedades simultáneamente, si todas ellas son disjuntas, se tendrán $k-2h+h = k-h$ caracteres para acomodar. Por otro lado, si dentro de las h parejas tomadas, existen l que compartan un dígito, dispondremos de $k-h+l$ elementos que acomodar, sin embargo, a esta expresión hay que restarle l (el número de propiedades que se omiten al unir los pares disjuntos). Por tanto, sin importar cuántos caracteres compartan los pares tomados, tendremos $k-h+l-l = k-h$ dígitos para colocar en $k-h$ lugares. Tomando en cuenta que se pueden elegir $\binom{k-1}{h}$ subconjuntos de h propiedades, $S_h = \binom{k-1}{h} \cdot (k-h)!$.

Así, la expresión que resuelve el problema es:

$$\sum_{i=1}^{k-1}(-1)^{i+1}\binom{k-1}{i}(k-i)! \tag{4.1}$$

Pero cada elemento de esa suma se puede escribir de la siguiente forma:

$$(-1)^{i+1}\binom{k-1}{i}(k-i)! = (-1)^{i+1}\cdot\frac{(k-1)!}{i!\cdot(k-1-i)!}\cdot(k-i)!$$

$$= (-1)^{i+1}\cdot\frac{(k-1)!}{i!}\cdot\frac{(k-i)\cancel{(k-i-1)!}}{\cancel{(k-i-1)!}}$$

$$= (-1)^{i+1}\cdot\frac{(k-1)!\cdot(k-i)}{i!}$$

Por lo que la suma 4.1, se puede reescribir como:

$$(k-1)!\cdot\sum_{i=1}^{k-1}(-1)^{i+1}\cdot\frac{k-i}{i!}$$

Finalmente, ya que es necesario aplicar módulos a la expresión y que hay una división presente, tendremos que recurrir a un procedimiento de *factorial reverso*, que podrás encontrar en la subsección *Divisiones en Aritmética Modular* de esta obra.

Caminos y más Caminos

VII Concurso Interno de Programación: Problema C

Schnitzel es una hámster muy inteligente y con mucho tiempo libre. Hace apenas unos días se mudó a un nuevo hogar que aún no termina de explorar, sin embargo, sabe que en él hay dos ruedas (A y B) entre las cuales hay un camino que aún desconoce; también sabe que los pasillos de su nuevo hogar tienen la forma de una cuadrícula de $m \times n$ y que A se encuentra en el extremo inferior izquierdo y B en el extremo superior derecho. Schnitzel quiere recorrer todos los posibles caminos que

hay para llegar de la primera a la segunda rueda.

Tu misión es ayudar a Schnitzel a calcular el tiempo que le tomaría pasar por todos los caminos posibles de A a B siguiendo las siguientes reglas:

1. Schnitzel siempre comienza su exploración en la rueda A y termina en la B.
2. Recorrer el lado de una celda cualesquiera le toma t segundos.
3. Únicamente puede avanzar hacia la derecha y hacia arriba.
4. Una vez que llegue a la segunda rueda, Schnitzel volverá a la primera por el camino que ella desee, y dado que este punto no cumple con la regla anterior, este camino no deberá ser considerado como uno más, así como el tiempo que tarde en hacerlo no deberá ser sumado a la duración de su paseo.

> **Solución.** En el capítulo de *Análisis Combinatorio*, aprendimos que podemos ver el avance desde A hacia B como una serie de decisiones en donde, en cada paso, elegiremos si ir hacia la derecha o hacia arriba, y sin importar qué orden tomen esas decisiones, Schnitzel invariablemente recorrerá $m+n$ celdas, m de las cuales serán hacia la derecha y n hacia arriba, por lo que, si contamos los momentos en los que decidimos ir hacia la derecha, los momentos en que vamos hacia arriba estarán dados. Así pues, existen
>
> $$\binom{m+n}{m}$$
>
> maneras de ir de A a B.
>
> Luego, hay que recordar que Schnitzel tardará t segundos en recorrer cada celda, por lo que el tiempo total para explorar cada camino será $t \cdot (m+n)$; finalmente, el tiempo que toma recorrer todos los caminos posibles de A a B es:
>
> $$t \cdot (m+n) \cdot \binom{m+n}{m}$$

Fibonaccis y Malvaviscos
VII Concurso Interno de Programación: Problema F

Malva es un malvavisco parlante al que le obsesionan los juegos con los números, y a quien le encanta ponerle retos a sus amiguitos. Cuando estos logran resolver sus problemas, les regala un iPad, pero cuando no lo logran, se burla de ellos.

Este reto consiste en lo siguiente: Malva dice un número natural n ($1 \leq n \leq 10^6$). Entonces tú debes obtener el n-ésimo término de la serie de Fibonacci. Pero esto no acaba aquí. Malva nunca lo hace fácil (porque no cuenta con tantos recursos para comprar tantos iPad). Una vez obtenido el n-ésimo término, debes sumar los dígitos del mismo. Si el número resultante tiene más de un dígito, debes volver a sumar todos sus dígitos. Así, repites este proceso hasta que tu resultado sea únicamente de un dígito.

A Malva le gusta que sigan sus instrucciones al pie de la letra, pero eso no es importante para mí, quien le dirá a Malva si lograste o no pasar su reto, así que si usas un procedimiento distinto, tranquilo, él no lo sabrá. Por último, considera que $f_1 = 1$ y $f_2 = 1$.

> **Solución.** Cuando demostramos el criterio de divisibilidad de 9 (en el capítulo de *Teoría de Números*), vimos que todo entero es congruente con la suma de sus dígitos, módulo 9; además, en el ejemplo 64 resolvimos una situación muy similar a la presentada en este problema. De ahí, concluimos que obtener el dígito pedido se logra al aplicar módulo 9 al número original, con la reserva de que, si el resultado es 0, éste debe ser intercambiado por 9.
>
> Hay que mencionar que, ya que obtendremos los términos de Fibonacci a cada paso, la opción ideal es aplicar programación dinámica y almacenar los datos que se vayan obteniendo, sin olvidar que en todos los casos se aplicará módulo 9.

Ahorrando la Fatiga
Concurso de la XVI Semana de MAC: Problema A

Para este problema te ahorraré la fatiga de leer y te diré que simplemente tienes que jugar a algo muy sencillo: contar. El juego consiste en que digas cuántas palabras de x_i ($1 \leq x_i \leq 10^5$) letras se pueden formar con los caracteres a, b y c sin que haya dos a's juntas. Para este juego entendemos como palabra una cadena de caracteres, sin importar que tenga sentido o no en alguno de los muchos idiomas que conoces.

> **Solución.** Revisar la solución del ejemplo 147 abordado en la sección *Programación Dinámica* de esta obra.

Bases, Bases Everywhere
Concurso de la XVI Semana de MAC: Problema B

¿No les pasa que cuando se transportan de un lugar a otro buscan relaciones en todos los números que ven por ahí? A mí sí, y les quiero compartir mi obsesión.

Les voy a explicar cuál es la dinámica de hoy: Tomo un número y sumo sus dígitos, luego tomo el número resultante y sumo sus dígitos. Repito este proceso hasta obtener un sólo dígito.

Cuando me di cuenta de que era muy fácil comencé a buscar cadenas de números con letras, con el propósito de tener números en bases 11, 12, 13, 14, 15 y 16. A estos números les aplico el mismo proceso. Así, tú debes decirme cuál es el caracter que me quedó al final.

> **Solución.** En el problema 80 (sección *Criterios de Divisibilidad*), demostramos que, dado un número en base B y al transformarlo a base decimal, encontramos su raíz digital aplicándome módulo $B-1$, con la reserva de que, si el resultado del módulo es 0, la respuesta al problema será B. Finalmente, se regresa el número obtenido a la base en la que estemos trabajando para obtener un único caracter de salida.

Fiesta de Ganadores
Concurso de la XVI Semana de MAC: Problema F
Autor: Moroni Silverio Flores

Queremos hacer una fiesta para los ganadores del concurso de programación, pero aquí entre nos, no tenemos mucho dinero, por lo que queremos rentar la menor cantidad de sillas que sea posible. Cada persona invitada envía previamente el minuto en que llega y el minuto en que se va (a partir del inicio de la fiesta, es decir, la fiesta inicia en el minuto 0), y lo cumple porque todos somos muy puntuales. Nosotros te daremos esos datos y tú determinarás cuál es la mayor cantidad de gente que puede haber en algún momento en la fiesta. Tu misión es decirnos dicha cantidad para que nosotros rentemos las sillas.

> **Solución.** Para resolver este problema, tendremos que almacenar juntos todos los minutos en los que una persona llega y se va, cuidando de identificar qué tipo de dato es (de llegada o de salida); luego ordenaremos dichos momentos en orden ascendente y sumaremos 1 cuando una persona llegue, y restaremos 1 cuando una persona se vaya, sustituyendo en valor del máximo únicamente si encontramos un punto en el que el número de persona exceda el dato anterior.
>
> Podemos clasificar esta solución como un procedimiento *greedy*, pues recurrimos a un ordenamiento de datos y no nos resulta importante lo que haya ocurrido en el pasado.

Hendrixland
Concurso de la XVI Semana de MAC: Problema H

Mr. Malvavisco es un ser muy perezoso. Justo ahora está de vacaciones en Hendrixland, y quiere recorrer todo lo que su pereza y su tiempo le permitan. Para eso te ha pedido ayuda a ti.

Mr. Malvavisco tiene en su poder una lista de los lugares a los que le gustaría ir, representados, cada uno, con uno de los primeros V ($1 \leq V \leq 10^4$) números naturales

distinto, así como de las A ($1 \leq A \leq 10^5$) conexiones inmediatas (las aristas) que existen entre ellos.

Como ya dije, Mr. Malvavisco es muy flojo, así que no está dispuesto a perder su tiempo intentando llegar a un lugar que no es alcanzable desde alguno de los puntos de su lista, por lo que tú debes avisarle antes de que lo intente y se enoje con todos nosotros.

> **Solución.** Nota que lo que buscamos responder en este problema es si existe un camino entre cualesquiera dos puntos del grafo que representa a Hendrixland, es decir, queremos saber si el grafo es conexo.
>
> No existe clarificación de que el grafo con el que tratamos sea dirigido, por lo que asumiremos que, para cada arista uv dada, existirá tanto la relación (u, v) como la (v, u).
>
> Dicho lo anterior, el problema se resolverá aplicando, ya sea una *búsqueda en amplitud* o una *búsqueda en profundidad* para verificar que el grafo sea conexo. Puedes encontrar el algoritmo en la subsección *Aplicaciones de Búsquedas* de esta obra.

Cuando una Hámster Quiere Jugar

VIII Concurso Interno de Programación: Problema C
Autor: Moroni Silverio Flores

A mi hámster le encanta jugar. Hoy decidió salir al patio a excavar hoyos para matar el aburrimiento. Para su comodidad, decidí trazar líneas en el patio de tal forma que éste fuera una cuadrícula, y en cada uno de los cuadros debe haber una piedra en algún punto bajo tierra que no le permitirá a la roedora hacer un hoyo más profundo.

Mi hámster comienza en el cuadro que más le guste en ese momento y saca toda su tierra hasta encontrar la piedra. Luego, se mueve a alguno de los cuadros con los que comparte al menos un vértice y escarba hasta que se encuentre con la piedra de ese cuadro, o bien, hasta que alcance la misma profundidad con la que quedó el

cuadro anterior, lo que ocurra primero. La inteligente roedora continúa con este proceso hasta que se encuentre con una piedra a nivel del suelo, es decir, a profundidad 0.

Considerando que la hámster puede moverse y regresar a donde ella desee y que puede moverse de manera horizontal, vertical y diagonal en la dirección que quiera, ¿cuál es la máxima cantidad de tierra que puede sacar?

Considera que el animalito tiene un asistente que retira la tierra en cuanto hace un hoyo, así que no tienes que preocuparte por pensar qué sucederá una vez que la tierra esté fuera del cuadro.

> **Solución.** Recurriremos a una solución que se asemeja al *algoritmo de Dijkstra* (que podrás encontrar en la subsección de *Ruta de Menor Peso*), para lo que veremos a la cuadrícula como un grafo en el que cada celda representa un nodo y donde existe una arista entre todo par de celdas con al menos un vértice en común.
>
> Así, estando en cualquier cuadro de la casa de la hámster, debemos excavar la casilla contigua (entendemos como contiguas dos casillas que comparten al menos una esquina) que permita ir más profundo (en el algoritmo original de Dijkstra tomamos siempre el vértice que permita sumar el menor peso).
>
> Tomando el punto inicial, se escarba en los puntos adyacentes a él, y para cada iteración, se toma el punto en el cual se escarbó más; a su vez, analizamos cuál de los vecinos de este último permite ir más profundo. El procedimiento se repite hasta encontrar una piedra a nivel del suelo.

Déjalo a la Suerte

VIII Concurso Interno de Programación: Problema D

Vamos a jugar un juego de azar. Si quieres participar, debes adquirir una papeleta en la que podrás seleccionar enteros de entre los n primeros números naturales y registrarlos. Puedes seleccionar la cantidad de números que quieras, siempre y cuan-

do sea más de uno y menos de $n-1$.

Como suele suceder con este tipo de juegos, habrá un anfitrión que sacará de una urna una papeleta ganadora, y ganará la persona que tenga la papeleta con los mismos números seleccionados que los del anfitrión. Para ganar, únicamente es necesaria la coincidencia con los números del anfitrión, sin importar el orden.

Si podemos asegurar que no hay dos papeletas iguales jugando, ¿cuál es la probabilidad de que ganes si adquieres k papeletas?

Como la probabilidad puede ser muy muy pequeña, necesitamos que multipliques tu respuesta por 10^{15}. Tendrás un margen de error de 10^{-5}.

Solución. Lo primero es dejar claro que la probabilidad de tener una papeleta ganadora es:

$$\frac{k}{\text{Total de papeletas}}$$

Ya que k es un dato dado, debemos concentrarnos en obtener el total de papeletas que están en juego.

Nota que, ya que una papeleta puede componerse de t enteros, en donde $2 \leq t \leq n-2$, buscamos encontrar el total de los subconjuntos de tamaño $2, 3, \ldots, n-2$ que se pueden formar de $[n]$, esto es:

$$\binom{n}{2} + \binom{n}{3} + \cdots + \binom{n}{n-3} + \binom{n}{n-2} \tag{4.2}$$

En la sección de *Coeficientes Binomiales* aprendimos que:

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n$$

Por lo que podemos reescribir la suma 4.2 de la siguiente manera:

$$\binom{n}{2} + \binom{n}{3} + \cdots + \binom{n}{n-3} + \binom{n}{n-2}$$

$$= 2^n - \left[\binom{n}{0} + \binom{n}{1} + \binom{n}{n-1} + \binom{n}{n}\right]$$

En la sección de *Coeficientes Binomiales* también aprendimos que $\binom{n}{i} = \binom{n}{n-i}$, por lo que:

$$\binom{n}{0} + \binom{n}{n} = 2\binom{n}{0} \quad \text{y} \quad \binom{n}{2} + \binom{n}{n-2} = 2\binom{n}{2}$$

Luego,

$$2\binom{n}{0} = 2(1) = 2$$

Y:

$$2\binom{n}{2} = 2 \cdot \frac{n!}{2! \cdot (n-2)!}$$

$$= \cancel{2} \cdot \frac{n \cdot (n-1) \cdot \cancel{(n-2)!}}{\cancel{2} \cdot \cancel{(n-2)!}}$$

$$= n^2 - n$$

Así pues, el total de papeletas será:

$$2^n - \left[\binom{n}{0} + \binom{n}{1} + \binom{n}{n-1} + \binom{n}{n}\right]$$

$$= 2^n - (2 + n^2 - n)$$

Con lo que nuestra respuesta será:

$$\frac{k}{2^n - (2 + n^2 - n)} \cdot 10^{15}$$

Referencias

1. M. Mareš, "Fairness of time constraints," Abril 2011, [En línea]. Disponible: http://mj.ucw.cz/papers/fairness.pdf. [Acceso Junio 2019].

2. Codility, "Time complexity," [En línea]. Disponible: https://codility.com/media/train/1-TimeComplexity.pdf. [Acceso Octubre 2019].

3. M. Silverio-Flores, "Juez guapa," Agosto 2018, [En línea]. Disponible: https://juezguapa.com/. [Acceso Julio 2019].

4. icpc.foundation, "The icpc regional rules," [En línea]. Disponible: https://icpc.baylor.edu/regionals/rules. [Acceso Agosto 2019].

5. N. M. Josuttis, *The C++ standard library : a tutorial and reference.* Michigan: Peasrson Education, 2012.

6. R. A. Brualdi, *Introductory Combinatorics.* República de China: China Machine Press, 2009.

7. M. L. Pérez-Seguí, *Combinatoria.* Ciudad de México: Universidad Nacional Autónoma de México, 2014.

8. M.-L. Pérez-Seguí, *Teoría de Números.* Ciudad de México: Universidad Nacional Autónoma de México, 2014.

9. F. Zaldívar, *Introducción a la Teoría de Números.* Ciudad de México: Fondo de Cultura, 2012.

10. D. Joyner, R. Kreminski, and J. Turisco, *Applied Abstract Algebra*. Maryland: Johns Hopkins University Press, 2004.

11. P. Pollack, "Euler and the partial sums of the prime harmonic series," *Elemente der Mathematik*, vol. 70, pp. 13–20, Enero 2015.

12. P. Louridas, *Real-World Algorithms: A beginner's guide*. Londres: MIT Press, 2017.

13. C. Paar and J. Pelzl, *Understanding Cryptography*. Berlín: Springer, 2010.

14. J.-P. Aumasson, *Serious Cryptography*. Zúrich: No Starch Press, 2017.

15. A. Tomescu, "Rolling hash (rabin-karp algorithm)," Febrero 2011, [En línea]. Disponible: https://people.csail.mit.edu/alinush/6.006-spring-2014/rec06-rabin-karp-spring2011.pdf. [Acceso Junio 2019].

16. T. O. Foundation, "Euclidean algorithm - oeiswiki," Marzo 2018, [En línea]. Disponible: https://oeis.org/wiki/Euclidean_algorithm. [Acceso Mayo 2019].

17. J. M. Manzano, "Olimpiadas de matemáticas. página de preparación y problemas," [En línea]. Disponible: http://wpd.ugr.es/∼jmmanzano/preparacion/index.php. [Acceso Julio 2019].

18. F. Halim and S. Halim, *Competitive Programming*. Singapur: NUS Press, 2013.

19. F. Halim, "Uhunt :: Uva hunting," [En línea]. Disponible: https://uhunt.onlinejudge.org/. [Acceso Noviembre 2018].

20. J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*. Nueva York: North-Holland, 1982.

21. G. Back, "Problem b: Bumped," in *North America Qualifier 2017*. http://cs.ecs.baylor.edu/∼hamerly/icpc/qualifier_2017/naq2017_problemset.pdf: ACM-ICPC North America Qualification Contest, 2017, pp. 3–4.

22. cplusplus.com, "lower_bound - c++ references," [En línea]. Disponible: http://www.cplusplus.com/reference/algorithm/lower_bound/. [Acceso Junio 2019].

23. F. I. Schaposnik, "Problem h: Hotel rewards," in *Latin American Regional Contest 2016*. http://maratona.ime.usp.br/hist/2016/resultados/contest.pdf: ACM-ICPC Latin America, 2016, p. 10.

Referencias

24. U. de las Ciencias Informáticas, "Caribbean online judge," [En línea]. Disponible: http://coj.uci.cu/contest/cproblem.xhtml?pid=4257&cid=1701. [Acceso Agosto 2019].

25. HackerRank, "Hackerrank. environment and samples," [En línea]. Disponible: https://www.hackerrank.com/environment/languages. [Acceso Septiembre 2019].

26. G. L. McDowell, *Cracking the Coding Interview*. Palo Alto: CareerCup, 2016.

27. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Steinl, *Introduction to Algorithms*, 2nd ed. Massachusetts: MIT Press, 2009.