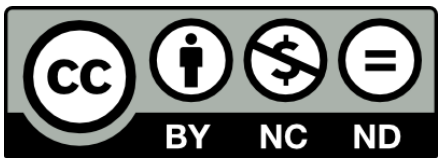
Algorítmica

Los Del DGIIM, `losdeldgiim.github.io`

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Algorítmica

Los Del DGIIM, losdeldgiim.github.io

Laura Mandow Fuentes

Granada, 2023-2024

Índice general

1. Introducción

Estos apuntes están enfocados a la *Programación Competitiva*. La programación competitiva se basa en el diseño y la implementación de algoritmos.

1.1. Tipos de problemas

Existen 3 tipos de problemas en programación competitiva:

- Input/Output
- Interactivos
- Output

En estos apuntes nos centraremos en los primeros.

1.2. Acortar código

En programación competitiva, también es muy importante la rapidez con la que se escribe el código. Por ello siempre se intenta tanto ahorrar escribir código como acortar el código escrito lo máximo posible.

Nota. Estas prácticas de programación van en detrimento de la legibilidad del código. No se aconseja su uso fuera del ámbito de la programación competitiva o de programas simples con el próposito exclusivo de probar algoritmos.

Typedefs

Gracias al comando `typedef` es posible renombrar los tipos de datos. De esta forma se les pueden otorgar nombres más cortos.

Algunos de los más comunes son:

```
typedef long long ll;
typedef long double ld;
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef pair<int,int> pi;
...
```

Otra forma de obtener el mismo resultado sería:

```
using ll = long long;
using ld = long double;
using vi = vector<int>;
using vvi = vector<vi>;
using pi = pair<int,int>;
...
```

Macros

Otra forma de acortar código es utilizar **macros**. Las macros son secciones de código que se sustituyen por otras antes de que el código sea compilado. Se definen usando *#define*.

Algunas de las macros más habituales en programación competitiva son:

```
#define F first
#define S second
#define pb push_back
#define mk make_pair
#define FOR(i,a,b) for(int i=a; i<b; ++i)
```

Un ejemplo de código empleando estos trucos sería:

```
vector<pi> v;
FOR(i,0,n){
    vi.pb(mk(i,2*i));
}
FOR(i,0,n){
    cout << v[i].F << " " << v[i].S  << endl;
}
```

Lo cual equivale a:

```
vector<pair<int,int>> v;
for(int i=0; i<n; ++i){
    vi.push_back({i,2*i});
}
for(int i=0; i<n; ++i){
    cout << v[i].first << " " << v[i].second  << endl;
}
```

Nota. Las macros pueden dar lugar a errores díficiles de encontrar. Abusar de ellas no es una buena praxis de programación.

Resaltamos que en el ejemplo anterior hemos empleando los corchetes `{}` en vez de la función `make_pair`. Ambos realizan la misma función. Se recomienda usar los primeros por acortar código.

Plantilla Básica

De ahora en adelante se entiende que en todos los códigos trabajamos con la siguiente plantilla de base:

```
1   #include <bits/stdc++.h>
2
3   using namespace std;
4
5   typedef long long ll;
6   typedef long double ld;
7
8   int main(){
9       ios::sync_with_stdio(false);
10      cin.tie(0);
11
12      return 0;
13  }
```

La primera línea incluye una librería que comprende todas las librerías que puedas llegar a necesitar. Esto evita tener que escribir lo siguiente en todos tus códigos:

```
#include <vector>
#include <queue>
#include <algorithm>
#include <cmath>
#include <string>
#include <set>
...
```

La segunda línea include un espacio de nombres muy usado tanto para entrada y salida como para usar la *STL*. De esta forma en vez de tener que escribir `std::cout` o `std::vector` podemos escribir `cout` y `vector` respectivamente.

Las líneas 9 y 10 se recomienda usarlas para aumentar la rapidez de la entrada y la salida del programa. Aunque no es lo normal, existen algunos casos de problemas y jueces concretos en los que se ha "apurado" bastante con el tiempo límite y ciertos códigos que implementan la solución esperada dan **TLE** [1].

1.3. Hacks de programación

Debugging

1.4. Plataformas de Jueces Online

Tanto para afianzar conocimientos de algorítmica como para entrenar para olimpiadas de programación competitiva, es necesario practicar aplicando las diversas técnicas y algorimos aprendidos resolviendo problemas.

Para ello, son de gran utilidad los llamados *jueces online*. Los *jueces online* son plataformas que comprenden una selección de problemas de programación (normalmente de *programación competitiva*, pero no todas se restringen a ellos), y que además incluyen el llamado *juez*.

[1] Para mayor información sobre jueces online y sus veredictos veáse 1.4

La dinámica que siguen estas plataformas es la siguiente: escoges uno de los problemas disponibles (suelen estar clasificados por dificultad, técnica de resolución...) diseñas e implementas la solución al problema y finalmente "submites" (envías) tu solución al *juez* para que la evalúe.

El *juez* ejecutará tu programa para una gran cantidad de casos de prueba (a los cuales no siempre tendrás acceso) y comparará la respuesta dada por tu programa con la respuesta (o respuestas, en caso de que haya más de una solución posible) esperada. En función de estas comparaciones emitirá un veredicto u otro (se examinarán más adelante). Algunas de las plataformas más conocidas son:

- Codeforces: Además de problemas también organiza concursos online periódicamente.
- CSES: **Muy recomendable** especialmente cuando estas empezando, puesto que esta plataforma te indica los casos en los que ha fallado tu código y la solución esperada. Tiene también una selección de problemas muy conviene para cuando vas aprendiendo algoritmos y técnicas nuevas.
- Hackerrank: También tiene problemas más enfocados a estructuras de datos o lenguajes de programación.
- Acepta el reto: Plataforma española donde se suben problemas de concuros de ámbito español.
- Leetcode: Al igual que *CSES* aporta las casos en los que falla tu código. También tiene cursos de aprendizaje.
- DMOJ: Muchas olimpiadas las realizan usando este juez.

Existen muchos más, estos son solo unos pocos.

Restricciones del problema

Lo más normal es que los problemas ofrezcan una serie de restricciones (o *constraints*) sobre los datos de entrada (o *input* del problema). Es decir, si los datos de entrada son dos enteros, lo más habitual es que se indique el rango de valores que puede tomar cada uno.

Ejemplo 1.4.1. Sean los datos de entrada son dos enteros, llámense n y m. Entonces las restricciones podrían ser:

$$1 \leqslant n \leqslant 10^9$$
$$-10^9 \leqslant m \leqslant 10^9$$

Además los problemas vienen acompañados de unas restricciones de tiempo y memoria. Esto es un máximo de segundos que tiene tu programa para ejecutar cada caso de prueba y de cuanta memoria puede disponer.

La restricción de tiempo unida a las restricciones de los datos de entrada condicionan el orden de **eficiencia** que debe tener nuestro algoritmo.

La restricción de memoria es raro que se supere, pero puede ser empleada para obligar a usar un algoritmo específico para un problema en el cual se podría aumentar la eficiencia en base a crear grandes estructuras de datos donde almacenar mucha información.

Veredictos

Al "submitir" (enviar) un código a uno de los jueces previamente mencionados (u otro), te responderán con uno de los siguientes veredictos:

- **AC (Accepted):** Felicidades, has resuelto correctamente el problema.
- **WA (Wrong Answer):** El código ha dado la respuesta incorrecta en alguno de los casos de prueba. Dependiendo del juez, se te informará de cual es el caso o no.
- **RU (Runtime Error):** El código ha dado un error en tiempo de ejecución. Uno de los motivos podría ser que estas dividiendo por 0.
- **PE (Presentation Error):** El código no ha impreso la solución siguendo el formato especificado.
- **TLE (Time Limit Exceeded):** Se ha excedido el tiempo máximo otorgado para acabar de ejecutarse tu código en alguno de los casos de prueba. Esto significa que el algoritmo empleado no es lo suficientemente **eficiente**.
- **MLE (Memory Limit Exceeded):** Se ha excedido el uso de la cantidad de memoria máxima permitida. Esto puede ser porque tengas muchos vectores muy grandes.

1.5. Testing

cph (incluyendo extension) stressed testing

2. Programación Dinámica

La **programación dinámica** es una técnica basada en **evitar repetir cálculos**. Combina la eficiencia de los algoritmos voraces (*greedy*) con la optimalidad de la búsqueda completa (siempre da la solución correcta). Se puede aplicar a un problema si este se puede **dividir en subproblemas superpuestos**.

Denominamos subproblemas a los problemas de la misma clase pero cuya variable es menor que la de nuestro problema original. Por ejemplo, si nuestro problema consiste en sumar 4 números, los subproblemas serían sumar 3 y 2 números. En este caso tenemos que el subproblema de sumar 3 números engloba al de sumar 2 números.

Otro ejemplo sería sumar 4 números y multiplicar el resultado por 3 números. Entonces tendríamos que los subproblemas sumar 3 números y multiplicar el resultado por 3 números y sumar 4 números y multiplicar el resultado por 2 números se **superponen** ya que ambos engloban al subproblema de sumar 2 números y multiplicar el resultado por 2 números (pero ninguno de los dos engloba al otro).

La programación dinámica tiene dos usos:

- Encontrar la **solución óptima** (problema de mínimos o máximos)
- Contar el **número de soluciones** (número total de soluciones posibles)

2.1. Fibonacci

Juez: Fibonacci

Uno de los ejemplos más básicos y clásicos para entender los fundamentos de la programación dinámica es calcular el n-ésimo término de la **sucesión de Fibonacci**. Esta es una fórmula recursiva cuya expresión $\forall n \in \mathbb{N}_0$ es:

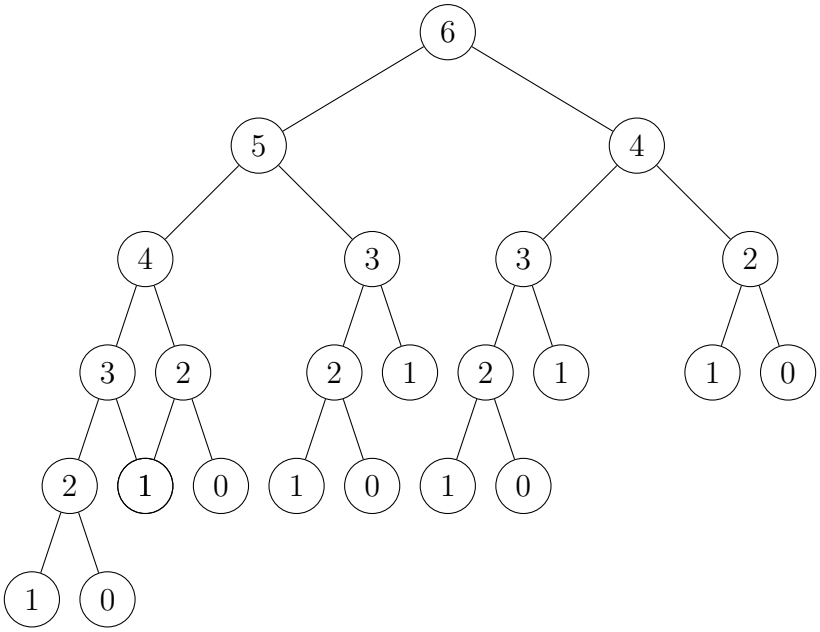
$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n > 1 \end{cases}$$

Pasada esta fórmula a código nos quedaría:

```
int f(int n){
    if(n == 0) return 0;
    if(n == 1) return 1;
    return f(n-1) + f(n-2);
}
```

Lo cual tiene una complejidad aproximada de $O(2^n)$, (de hecho es $O(\frac{1+\sqrt{5}}{2})$), es decir, **exponencial**.

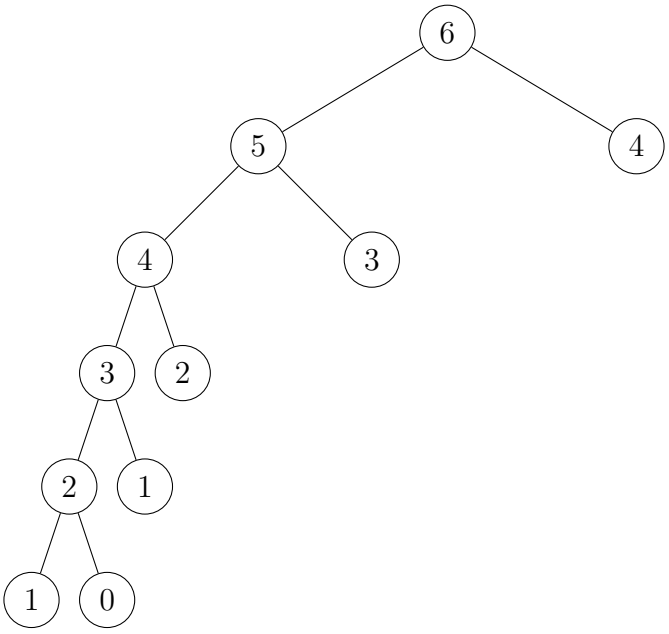
Es fácil observar que nuestro problema reside en que estamos repitiendo los mismo cálculos reiteradamente. Véamos el árbol de llamadas de la función recursiva para $n = 6$.



Podemos ver claramente que estamos calculando $n = 2$ nada menos que **5 veces**. Los subproblemas $n = 3$ y $n = 4$ también se calculan más de una vez. Además observamos que los subproblemas se solapan (superponen) entre sí. Lo ideal sería simplemente calcular cada subproblema **una única vez**.

"Aquellos que no recuerdan su pasado están condenados a repetirlo"[1]

De esta forma el árbol de llamadas quedaría de la siguiente forma:



[1]Frase dicha por el filósofo Jorge Agustín Nicolás Ruiz de Santayana y Borrás a la entrada del campo de exterminio nazi de Auschwitz.

Podemos ver que el número de llamadas realizadas se reduce drásticamente, sobre todo en este caso que pasamos del orden **exponencial** al orden **lineal** dado que cada término de la secuencia de Fibonacci se calcula **una única vez**.

Esta aplicación de la programación dinámica se llama **memoization**, y consiste simplemente en eso, "recordar" los cálculos ya realizados. Esto se implementa almacenando las soluciones de los **subproblemas** en una tabla (N-dimensional), que normalmente denominaremos **dp** (de *dynamic programming*).

Por tanto tenemos que nuestro código anterior pasa a ser:

```
const int MAXN = 100;
int dp[MAXN];
bool calculated[MAXN];

int f(int n){
    if(calculated[n]) return dp[n];
    if(n == 0) return 0;
    if(n == 1) return 1;

    calculated[n] = true;
    return dp[n] = f(n-1) + f(n-2);
}
```

Tenemos pues un array de `bool` inicializado a `false` que nos indica si hemos calculado ya el n-ésimo término de la sucesión o no. Si no lo hemos calculado, recurrimos a la fórmula previamente vista. En caso contrario, en vez de volver a calcularlo, simplemente accedemos al lugar en memoria en el que lo tenemos almacenado (en nuestro caso el array `dp`), ahorrándonos todas las llamadas recursivas que supondría recalcularlo. De esta forma, nos queda el árbol de llamadas ya visto, y por tanto la eficiencia de la función es claramente $O(n)$.

Otra forma de implementarlo sería:

```
const int MAXN = 100;
vector<int> dp(MAXN,-1);
dp[0] = 0;
dp[1] = 1;

int f(int n){
    if(dp[n] == -1)
        dp[n] = f(n-1) + f(n-2);
    return dp[n];
}
```

La principal diferencia frente al código anterior reside en que en vez de tener dos arrays, uno para saber si ya tenemos calculado el subproblema y otro con la solución correspondiente al subproblema, tenemos uno solo, que nos indica ambas cosas. Esto se debe al uso del centinela -1 (valor imposible) para representar que la solución al n-ésimo subproblema no ha sido calculada todavía.

Ambas implementaciones siguen el enfoque descendente (***top-down***), de forma que llamamos a la función para preguntarle por el valor de n y esta va desde "arriba (*top*)" que sería el valor por el que le hemos preguntando hasta "abajo (*down*)" que serían los casos bases de la recursión.

Existe otro enfoque, el ascendente (***bottom-up***). Este parte de los casos bases y los va extendiendo a casos más grandes. Véamoslo aplicado a la secuencia de Fibonacci.

```
const int MAXN = 100;
int fib[MAXN];

int f(int n){
    fib[0] = 0;
    fib[1] = 1;

    for(int i = 2; i ⩽ n; ++i){
        fib[i] = fib[i-1] + fib[i-2];
    }
}
```

Tenemos pues que la función nos calcula los n-ésimo primeros términos de la sucesión. No tiene mucho sentido llamar a la función varias veces para saber términos puntuales, ya que ahí sí que estaría repitiendo cálculos (recordamos además que la función sigue siendo $O(n)$). En caso de necesitar solo el término n-ésimo tendríamos que estaríamos derrochando memoria (usamos $O(n)$ de memoria) cuando solo nos hace falta recordar los 2 últimos términos de la sucesión para calcular el siguiente (ahora mismo almacenamos la sucesión entera hasta n). Para hacerlo más eficiente respecto a la memoria, una opción sería:

```
const int MAX_SAVE = 3;
int fib[MAX_SAVE];

int f(int n){
    fib[0] = 0;
    fib[1] = 1;

    for(int i = 2; i ⩽ n; ++i){
        fib[i % MAX_SAVE] = fib[(i-1) % MAX_SAVE] + fib[(i-2
    ) % MAX_SAVE];
    }
}
```

Aplicando módulo de la constante `MAX_SAVE` que almacena el número de términos previos de la sucesión que tenemos almacenados, solucionamos nuestros problemas de espacio, pues pasamos de usar $O(n)$ a $O(1)$.

2.2. Devolver cambio

Problema

Juez: Devoler Cambio

El problema de devolverle cambio a un cliente se traduce por: dado un conjunto de valores de monedas llamémosle `monedas` (ej: 1,2,5,10,20...) y un valor x, queremos sumar el valor x pedido mediante las monedas dadas usando el **mínimo** número de monedas posibles (se tienen infinitas monedas de cada valor). Destacamos que tanto los valores que pueden tomar las monedas como el cambio devolver son números naturales. En resumidas cuentas tenemos:

- monedas de n valores *naturales*
- suministro ilimitado de monedas

- debemos formar la suma x con los valores disponibles
- debemos usar el **mínimo** número de monedas posibles

Este problema puede resolverse mediante un algoritmo *greedy*, de escoger siempre la moneda de mayor valor posibles. Este enfoque nos da la solución óptima si el conjunto de monedas es por ejemplo el de los euros, pero en general no tenemos garantizado que nos de la solución óptima.

Ejemplo 2.2.1. Supongamos que disponemos de infinitas monedas de valores 1,4,6 y queremos devolver 8.

El algoritmo greedy nos devolvería:

- $6 \leqslant 8 \implies$ una moneda de 6, me resta por devolver 2.
- $4 > 2, 1 \leqslant 2 \implies$ una moneda de 1, faltan 1 por devolver
- $1 \leqslant 1 \implies$ una moneda de 1

Por tanto nos devolvería **3** monedas (1 de 6 y 2 de 1), mientras que la solución óptima sería devolver **2** monedas de 4.

Procedamos ahora a resolver el problema aplicando programación dinámica. Para ello basaremos nuestro enfoque en una función recursiva que explora todas las posibles combinaciones para formar la suma, como si fuera *fuerza bruta*. No obstante al aplicar programación dinámica, en particular lo que hemos llamado **memoization** nos resultará más eficiente ya que calcularemos la solución a cada subproblema una **única** vez.

Enfoque básico

Definición recursiva

La programación dinámica se basa en formular la solución al problema recursivamente de forma que la solución se pueda calcular a partir de la soluciones de los subproblemas.

Supongamos que trabajamos con los valores de monedas del ejemplo 2.2.1, es decir, `monedas = {1,4,6}`. Llamémos `cambio(x)` a nuestra función recursiva que nos devuelve el mínimo número de monedas necesarias para sumar x.

La idea clave consiste en fijarnos en la **primera** moneda que escogemos. Supongamos $x = 8$. En este caso puede ser 1,4 o 6. Si escogemos la moneda 1, nuestro nuevo problema ahora consiste en calcular el mínimo número de monedas necesarias para sumar $x-1$, es decir 7, lo cual es un subproblema de nuestro problema original. De igual forma se haría de haber escogido 4 o 6. A partir de este solución parcial podemos obtener la solución a nuestro problema original (simplemente habría que sumar 1 por la primera moneda que escogimos). Tenemos pues que nuestra función puede definirse de la siguiente forma:

```
cambio(x) = min(cambio(x-1)+1,cambio(x-4)+1,cambio(x-6)+1)
```

Obviamente habría que comprobar que podemos escoger una moneda, es decir, si $x = 5$, entonces claramente no podemos escoger la moneda 6 como primera moneda.

El caso base de la recursión sería `cambio(0) = 0` dado que si no hay que devolver cambio entonces no devolvemos ninguna moneda. Formalmente tendríamos:

$$cambio(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min\limits_{c \in monedas} cambio(x - c) + 1 & x > 0 \end{cases}$$

Para $x < 0$ tenemos que el valor es ∞, es decir un valor imposible ya que no podemos formar sumas negativas con valores positivos. En este caso hemos escogido ∞ (a la hora de implementarlo a código esto sería un número inalcanzable debido a las restricciones del problema) ya que se trata de un problema mínimos y de esta forma al asociarle un valor muy grande evitamos escoger una opción imposible. En el ejemplo anterior, tendríamos que `cambio(x-6)` nos daría infinito y por tanto la función mínimo lo "ignorará" y se quedará con los valores dados por `cambio(x-4)` o `cambio(x-1)` (independientemente de lo que devuelvan estas funciones siempre será menor que "infinito"). Además sirve para representar que no existe ninguna solución posible en caso de que no se pueda formar la suma con los valores de monedas dados. Por ejemplo no podemos formar la suma 5 con monedas de 2 y 4.

Para $x = 0$ hemos visto que no necesitamos devolver nada.

Por último para $x > 0$ iteramos por todos los valores de monedas posibles y los quedamos con el mínimo de escoger como primera moneda todas ellas + 1 (por la moneda inicial escogida).

Pasando esta definición a código (donde `INF` representa un valor muy muy alto e imposible de alcanzar según las restricciones del problema), nos queda:

```
int change(int x){
    if(x < 0) return INF;
    if(x == 0) return 0;
    int sol = INF;
    for(int c : coins){
        sol = min(sol,change(x-c)+1);
    }
    return sol;
}
```

Aplicando memoization

Esta sería la implementación de la solución de fuerza bruta la cual podemos constatar que es **muy ineficiente**. Procedamos ahora a aplicarle programación dinámica, en particular, **memoization** para aumentar su eficiencia.

```
vector<int> dp(W+1,-1);
int change(int x){
    if(x < 0) return INF;
    if(x == 0) return 0;
    if(dp[x] == -1){
        dp[x] = INF;
        for(int c : coins){
            dp[x] = min(dp[x],change(x-c)+1);
        }
    }
    return dp[x];
}
```

Donde hemos notado por W a una constante mayor que todos los tamaños de problemas que nos interesan y en `dp` almacenamos los resultados de los subproblemas ya calculados, usando -1 como centinela para saber si ya hemos resuelto ese subproblema o no (en ningún caso podemos devolver -1 monedas). La complejidad pasa de ser exponencial a $O(nx)$ donde n es el número de valores de monedas disponibles y x el cambio que queremos devolver.

Exactamente el mismo algoritmo puede implementarse iterativamente (**bottom-up**).

```
vector<int> dp(W+1,INF);
dp[0] = 0;
for(int x = 1; x <= W; ++x){
    for(int c : coins){
        if(x >= c){
            dp[x] = min(dp[x],dp[x-c]+1);
        }
    }
}
```

Enfoque general

Veámos ahora otra forma de resolver el mismo problema con programación dinámica, pero de una forma más "general" o parecida a como se suelen resolver los problemas aplicando esta técnica. En nuestro algoritmo inicial, llamémoslo "básico" empleando un vector `dp` **unidimensional**, dado que nos hemos preocupado solo por el cambio que queríamos devolver y no tanto por el número de valores de monedas del que disponemos. Nos hemos podido despreocupar de esto debido a la simplicidad de nuestro problema, pero esto no siempre (de hecho en general) no será así.

Examinemos ahora el problema desde otra perspectiva más instructiva, donde nuestra función recursiva `cambio` no solo depende de x sino que también depende de k, donde k representa que tan solo disponemos de los k primeros valores de `monedas`. Tendremos pues que usar un vector `dp` **bidimensional** para controlar ambas variables.

Definición recursiva

Nuestra función recursiva `cambio(k,x)` depende ahora de dos variables. Si n es el número de valores de monedas disponibles tenemos $k \in [0, n]$ (ya que no tiene

sentido considerar los -2 primeros elementos ni más de n primeros elementos). Su definición recursiva $\forall x, k \in \mathbb{N}_0$ es:

$$cambio(k,x) = \begin{cases} \infty & x < 0 \vee (k = 0 \wedge x \neq 0) \\ 0 & x = 0 \\ min(cambio(k-1,x), cambio(k, x - c_k) + 1) & x > 0 \wedge k > 0 \end{cases}$$

Donde denotamos por c_k al k-ésimo valor de moneda. Los casos bases de la recursión siguen siendo los mismos de antes ($x \leqslant 0$) al cuál añadimos uno más, que es $k = 0$, en cuyo caso devolvemos ∞ (salvo que $x = 0$). Esto tiene sentido ya que si no disponemos de monedas se nos es **imposible** devolver el cambio (suma) estipulado (salvo que no haya que devolver nada).

Además tenemos que el caso general ha sido modificado (como era de esperarse al haber cambiado los parámetros de la función). Analicemos la nueva fórmula dada. Queremos formar la suma x con los k primeros valores de monedas dados con el mínimo número de monedas posibles. Tenemos todos los subproblemas resueltos para valores menores tanto de x como de k, para cualquier combinación. En particular, tenemos resuelto el subproblema de formar la suma x para los $k-1$ primeros elementos. La única diferencia entre este problema y el nuestro, es que ahora disponemos de un valor de moneda más. Por tanto nuestro verdadero problema reside en **si nos conviene usar este nuevo valor o no**.

En el caso de que **no** nos convenga usar este nuevo valor (no forma parte de la solución del problema) o simplemente no podamos ($c_k > x$) es obvio que la solución es la misma que para los $k-1$ primeros valores. Es decir, en ese caso tenemos `cambio(k,x) = cambio(k-1,x)`. Por tanto, simplemente no usamos monedas de valor c_k.

En caso contrario tenemos que usar monedas de valor c_k **sí** que nos conviene (mejora la solución con tan solo los $k-1$ primeros valores de monedas). Por tanto, nuestra primera moneda será de valor c_k y nos queda un subproblema de la misma clase en el que queremos conocer el mínimo número de monedas necesarias para formar la suma $x - c_k$ usando los k primeros valores de monedas (nuestra solución es esta sumando 1 por la moneda inicial).

En resumidas cuentas, para formar la suma x con los k primeros elementos en principio tenemos una solución usando solo los $k-1$ primeros elementos. A partir de ahí esta solución puede ser mejorada al usar el k-ésimo elemento (o no y nos quedaríamos con la solución que ya teníamos antes).

Ejemplo 2.2.2. Supongamos que tenemos que devolver de cambio 8 y disponemos de 3 valores de monedas que son 1,4 y 6. Es decir, $x = 8, n = 3, monedas = \{1, 4, 6\}$.

Rellenemos la tabla donde almacenamos las soluciones para cada subcaso constructivamente.

k / x	0	1	2	3	4	5	6	7	8
$k = 0$	∞	∞	∞	∞	∞	∞	∞	∞	∞
$c_1 = 1$	0								
$c_2 = 4$	0								
$c_3 = 6$	0								

A partir de estos subcasos, aplicando la fórmula dada podemos ir rellenando la tabla. Comencemos por la segunda fila en la cuál podemos usar solo monedas de valor 1. Tenemos pues:

k / x	0	1	2	3	4	5	6	7	8
$k = 0$	0	∞	∞	∞	∞	∞	∞	∞	∞
$c_1 = 1$	0	1	2	3	4	5	6	7	8
$c_2 = 4$	0								
$c_3 = 6$	0								

Esto es evidente dado que al solo tener monedas de 1 el número mínimo de monedas se corresponde con x. En este caso en particular siempre nos quedamos con el segundo término del mínimo ($cambio(k, x - c_k) + 1$) dado que la fila 0 vale entera ∞. Procedamos con la segunda fila.

k / x	0	1	2	3	4	5	6	7	8
$k = 0$	0	∞	∞	∞	∞	∞	∞	∞	∞
$c_1 = 1$	0	1	2	3	4	5	6	7	8
$c_2 = 4$	0	1	2	3	1	2	3	4	2
$c_3 = 6$	0								

En este caso tenemos que para los $x < 4$ no podemos usar monedas de valor 4, pero a partir de este sí y de hecho nos resulta ventajoso mejorando las soluciones de la fila anterior. Por último la tabla nos queda:

k / x	0	1	2	3	4	5	6	7	8
$k = 0$	0	∞	∞	∞	∞	∞	∞	∞	∞
$c_1 = 1$	0	1	2	3	4	5	6	7	8
$c_2 = 4$	0	1	2	3	1	2	3	4	2
$c_3 = 6$	0	1	2	3	1	2	1	2	2

Tenemos pues que la última fila mejora algunos subproblemas pero en particular **no** mejora el caso final de $x = 8$ puesto que la solución óptima no usa monedas de 6 (sino dos de 4).

Es el ejemplo que vimos antes. Tenemos que la solución óptima para sumar 8 con monedas de 1 y de 4 es usar 2 monedas de 4. Procedemos a preguntarnos si podemos mejorar esta solución (2) usando (además) monedas de 6. Tenemos que al usar monedas de 6 necesitamos un total de 3 monedas (1 de 6 y 2 de 1), lo que empeora nuestra solución anterior, por tanto nos quedamos con la anterior.

En el caso de sumar 7 ocurre lo contrario. Tenemos que la solución para monedas de 1 y de 4 es de 4 monedas (1 de 4 y 3 de 1). Si además usamos monedas de 6, solo debemos usar 2 monedas (1 de 6 y 1 de 1) lo que mejora la solución anterior, por lo que nos quedamos con esta.

Denotando por N a el número de valores de monedas disponibles, por INF a un valor imposiblemente alto, por W a el máximo cambio a devolver y considerando que `monedas` empieza en la posición 1 (la posición 0 es valor basura y no se usa) la implementación recursiva del código sería:

```
vector<vector<int>> dp(N+1,vector<int> (W+1,-1));
int change(int k, int x){
    if(x == 0) return 0;
    if(x < 0 || k == 0) return INF;
    if(dp[k][x] == -1){
        if(x >= coins[k])
            dp[k][x] = min(cambio(k-1,x),
                           cambio(k,x-coins[k])+1);
        else
            dp[k][x] = cambio(k-1,x);
    }
    return dp[k][x];
}
```

Equivalentemente iterativamente, de forma que la solución se va construyendo poco a poco sería:

```
int dp[N+1][W+1];

// Caso base k = 0 (fila 0)
for(int x = 1; x < W+1; ++x){
    dp[0][x] = INF;
}

// Caso base x = 0 (columna 0)
for (int k = 0; k < N+1; k++){
    dp[k][0] = 0;
}

// Caso general
for (int k = 1; k < N+1; k++){
    for(int x = 1; x < W+1; ++x){
        dp[k][x] = dp[k-1][x];
        if(x >= coins[k])
            dp[k][x] = min(dp[k][x],dp[k][x-coins[k]]);
    }
}
```

Construyendo la solución

Puede ser que en algún caso no nos pidan solo el valor la solución óptima sino que además nos pidan un ejemplo de como construir dicha solución.

En el caso del enfoque básico de "Devolver cambio", podemos simplemente declarar un array `first_coin` para ir almacenando el valor de la primera moneda escogida de la solución óptima. De esta forma si para $x = 8$ sabemos que la primera moneda escogida fue `first_coin[x = 8] = 4` entonces la siguiente moneda escogida sabemos que es `first_coin[8 - 4 = 4] = 4` y así hasta llegar a $x = 0$.

Por tanto el código para calcular **una** solución (las soluciones no tienen porque ser únicas) e imprimirla con el enfoque básico quedaría así:

```
vector<int> dp(W+1,INF),first_coin(W+1);
dp[0] = 0;
for(int x = 1; x <= W; ++x){
    for(int c : coins){
        if(x >= c && dp[x-c]+1 < dp[x]){
            dp[x] = dp[x-c]+1;
            first_coin[x] = c;
        }
    }
}

// Construir sol
while(x > 0){
    cout << first_coint[x] << " ";
    x -= first_coin[x];
}
```

Por otro lado, si nos fijamos ahora en el enfoque general, vemos que ya tenemos toda la información necesaria para construir la solución gragias a nuestra tabla `dp`. Esto se debe a que podemos reconstruir nuestros pasos de cuando la rellenamos.

Recordamos que al rellenar la tabla, en particular al determinar el valor de `dp[k][x]` (sumar x con monedas de los k primero valores), teníamos dos opciones, usar una moneda de valor c_k o no usarla. Podemos saber el resultado de este decisión en todos los casos haciendo simples comparaciones. Si decidimos no usar como primera moneda una de valor c_k, tendremos que `dp[k][x] = dp[k-1][x]` y podemos repetir así el proceso hasta dar con las monedas usadas. En caso de no cumplirse esta igualdad, es decir, haber usado como primera moneda una de valor c_k repetimos el mismo procedimiento que para el enfoque básico.

Recordando que nuestro array de `coins` empieza en 1 el código nos quedaría:

```
int k = n;
while(x > 0){
    while(k > 1 && dp[k][x] == dp[k-1][x])
        --k;
    cout << coins[k] << " ";
    x -= coins[k];
}
```

Contando el número de soluciones

Juez: Devolver cambio (contar soluciones)

Por otro lado, como mencionamos antes, la programación dinámica no solo nos sirve para resolver problemas de optimización, sino que también para **contar el número** de soluciones de un problema.

Supongamos que ahora los que nos piden no es hallar la solución óptima (mínimo número de monedas necesarias) ni construirla para el problema de "Devoler cambio", sino que nos piden contar el número total de soluciones. Es decir, de cuantas formas distintas podemos sumar x (devoler el cambio) con los valores de monedas dados.

Ejemplo 2.2.3. Retomemos nuestro ejemplo original 2.2.1 en el cual queremos devolver $x = 8$ y disponemos de monedas de 1,4 y 6. El número de formas distintas de devolver el cambio son 10:

- 1+1+1+1+1+1+1+1
- 4+1+1+1+1
- 1+4+1+1+1
- 1+1+4+1+1
- 1+1+1+4+1
- 1+1+1+1+4
- 4+4
- 6+1+1
- 1+6+1
- 1+1+6

Tenemos pues que el razonamiento es muy similar al seguido en el problema original. Primero escogemos la primera moneda que puede ser de uno de los n valores dados (c_k) y nos queda un problema de la misma clase pero para $x - c_k$. Solo que en este caso en vez de quedarnos con el mínimo de las n opciones (al buscar la solución óptima), las **sumamos** (porque queremos contar **todas** las soluciones posibles).

Definición recursiva

Formulemos de nuevo esta idea recursivamente. Supongamos que trabajamos con los valores de monedas del ejemplo 2.2.1, es decir, `monedas = {1,4,6}`. Llamémos `contarSols(x)` a nuestra función recursiva que nos devuelve el número total de formas que se pueden sumar x.

La idea clave vuelve a ser en fijarnos en la **primera** moneda que escogemos. Supongamos $x = 8$. En este caso puede ser 1,4 o 6. Si escogemos la moneda 1, nuestro nuevo problema ahora consiste en calcular el número total de formas que se pueden sumar $x - 1$, es decir 7, lo cual es un subproblema de nuestro problema original. De igual forma se haría de haber escogido 4 o 6. A partir de este solución parcial podemos obtener la solución a nuestro problema original (simplemente habría que sumar los resulados de todas las soluciones parciales para así contar todas las formas posibles de sumar x con los valores de monedas dados). Tenemos pues que nuestra función puede definirse de la siguiente forma:

```
contarSols(x) = contarSols(x-1) + contarSols(x-4)
              + contarSols(x-6)
```

Obviamente habría que comprobar que podemos escoger una moneda, es decir, si $x = 5$, entonces claramente no podemos escoger la moneda 6 como primera moneda.

El caso base de la recurrencia sería `contarSols(0) = 1` dado que si no hay que devolver (sumar) nada solo hay una forma de hacerlo (devolver 0). En el caso de no poder formarse la suma de ninguna forma, tenemos que el número de soluciones es 0. Formalmente tendríamos:

$$contarSols(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in monedas} contarSols(x - c) & x > 0 \end{cases}$$

Pasando esta definición a código (iterativamente, **bottom-up**), nos queda:

```
1 vector<int> dp(W+1,0);
2 dp[0] = 1;
3 for(int x = 1; x ⩽ W; ++x){
4     for(int c : coins){
5         if(x >= c){
6             dp[x] += dp[x-c];
7         }
8     }
9 }
```

A veces el número total de soluciones es exageradamente alto (demasiado para el ordenador, daría *overflow*). En estos casos no se suele pedir el número exacto sino su módulo *mod* (normalmente $mod = 10^9 + 7$). Para el código anterior, simplemente añadimos la siguiente línea de código después de la línea 6:

```
1     dp[x] %= mod;
```

Se sugiere tras haber comprendido está sección intentar el ejercicio 2.5.1 Devolver cambio (contar soluciones *ordenadas*). Es exactamente el mismo problema solo que en este caso las soluciones 1+1+6 y 1+6+1 son iguales (no importa el orden en el que se dan las monedas).

Pista: Compara los dos enfoques aplicados a este problema (básico y general), en particular a la versión en la que hay que *construir* una solución óptima.

2.3. Mochila

Juez: Knapsack I

Veamos ahora el problema probablemente más famoso de programación dinámica. El problema de la mochila (*Knapsack* en inglés). Existen numerosas versiones de este problema, una de ellas es el de la *mochila fraccional*, que ya vimos que se resolvía mediante un algoritmo *greedy*. Pero obviando esta notable excepción, en general todas las demás variantes de este problema se resuelven con programación dinámica. Realmente se les llama *Knapsack problems* a aquellos problemas en los que te dan un conjunto de objetos y se deben encontrar subconjuntos que cumplan una serie de requisitos.

Centrémonos en el problema clásico: tenemos una mochila (como no) de capacidad máxima W y un conjunto de n objetos de peso w_i y un valor c_i. Y queremos llevar el máximo valor posible que nos quepa en la mochila.

Por tanto, tenemos que seleccionar los elementos que nos proporcionen un beneficio máximo pero cuyo peso global sea menor o igual que W (no se pueden partir los objetos).

Definición recursiva

El razonamiento que seguiremos para resolver este problema es similar el enfoque general del problema de "Devolver cambio" de la sección 2.2. Nuestro problema se resume en si nos conviene coger el objeto k-ésimo o no. Llamemos `mochila(k,x)` a la función que nos devuelve el máximo beneficio considerando los k primeros objetos y una mochila de capacidad x. Tenemos pues que la solución a este subproblema es el máximo entre no coger el elemento k, es decir, la solución para una mochila

de capacidad x con los $k-1$ primeros elementos y coger el elemento k, que implica el valor del k-ésimo elemento sumado a la solución para una mochila de capacidad $x - w_k$ (le quitamos el espacio que nos ocupa el elemento k) para los $k-1$ primeros elementos (no podemos repetir elementos, es decir, hay un elemento de cada).

$$mochila(k,x) = \begin{cases} 0 & x = 0 \vee k = 0 \\ max(mochila(k-1,x), mochila(k-1,x-w_k) + c_k) & x > 0 \wedge k > 0 \end{cases}$$

Obviamente si consideramos 0 elementos o una mochila de capacidad 0 nuestro máximo beneficio posible será 0.

Ejemplo 2.3.1. Supongamos que tenemos una mochila de capacidad $W = 5$, y $n = 4$ objetos tales que:

item	peso	valor
1	2	12
2	1	10
3	3	20
4	2	15

Rellenemos la tabla `dp` donde almacenamos las soluciones para cada subcaso constructivamente. Primero los casos base.

k / x	0	1	2	3	4	5
k = 0	0	0	0	0	0	0
$w_1 = 2, c_1 = 12$	0					
2	0					
3	0					
4	0					

Se sugiere tras haber comprendido está sección intentar el ejercicio 2.5.2 Knapsack II. Es exactamente el mismo problema. Solo cambian las restricciones de los casos de entrada (recordar que un vector de más de 10^8 celdas **nunca** es una buena idea).

2.4. Subsecuencia de longitud mayor (LCS)

2.5. Problemas

Aplicamos ahora la nueva técnica aprendida a la resolución de diversos problemas. Se aconseja intentar estos problemas antes de mirar la solución. En todos los casos se ha facilitado el juez correspondiente para que se pueda enviar la solución y comprobar si esta correcta o no. Recordamos las soluciones de las problemas de programación **nunca** son únicas (aunque se pueden parecer) y no debe preocuparle que su solución no se parezca a la sugerida siempre y cuando el juez la acepte, es decir, el veredicto sea Accepted (AC).

Consejo: Si se atasca con un problema, no se desespere. Dese una vuelta o despéseje de alguna forma. En caso de seguir resistiéndose el problema, no pierda más tiempo y mire la solución.

Ejercicio 2.5.1. Devolver cambio (contar soluciones *ordenadas*).

Enunciado: Dados un **entero** x y un array de n valores (enteros también), calcular el número de formas **ordenadas** distintas con el que se puede sumar x usando los n valores dados. Ponemos énfasis en que 1+2 y 2+1 constituirían la misma solución y contarían **una única** vez. (para $x = 3$ y el array = 1,2)

Solución:

Una versión distinta de este problema ya fue explicada en la sección 2.2, cuya diferencia residía en que no se podían las formas **ordenadas** sino **todas**. Es decir, 1+2 era una forma distinta a 2+1. En ese problema utilizamos simplemente un array *unidimensional*. Esto no fue casualidad, y es que en ese array solo estamos teniendo en cuenta el valor de x de forma que no tenemos en cuenta el orden de los valores de monedas proporcionados de ninguna forma. Es decir, para toda la tabla usamos todos los valores de moneda indistintamente.

En cambio en esta versión **sí** que tenemos que tener en cuenta el orden para evitar contar una misma solución más de una vez. Para tener en cuenta esto le añadimos una dimensión más a nuestra tabla (tal y como vimos en el enfoque general) para tener en cuenta los valores de monedas usados para formar la solución. Tenemos pues una tabla `dp` *bidimensional* donde `dp[k][x]` guarda el número de formas **ordenadas** de sumar x con los k primeros valores dados. Entonces tenemos que el orden **sí** que se está teniendo en cuenta, puesto que la primera moneda escogida siempre será la de una posición mayor en el array de sus valores (la k-ésima en caso de haberla escogido). Es decir, para calcular `dp[k][x]` sumo las formas de calcular x con los $k-1$ primeros elementos con las formar de sumar $x - c_k$ (si $x \geqslant c_k$), siendo c_k el valor de la primera moneda.

Implementación:

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef long double ld;

// MOD
const ll mod = 1e9 + 7;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);

    // INPUT
    ll n,x;
    cin >> n >> x;

    int coins[n];
    for(int i = 0; i < n; ++i){
        cin >> coins[i];
    }

    // SOLVE
    int dp[n+1][x+1];

    // Caso base k = 0 (x != 0)
```

```
    for(int i = 1; i <= x; ++i){
        dp[0][i] = 0;
    }

    // Caso base x = 0
    for(int k = 0; k <= n; ++k){
        dp[k][0] = 1;
    }

    // Caso general
    for(int k = 1; k <= n; ++k){
        for(int i = 1; i <= x; ++i){
            dp[k][i] = dp[k-1][i];
            if(coins[k-1] <= i){
                dp[k][i] += dp[k][i-coins[k-1]];
                dp[k][i] %= mod;
            }
        }
    }

    // OUTPUT
    cout << dp[n][x] << endl;

    return 0;
}
```

No obstante, está implementación tiene un serio problema de memoria, y es que para valores de x y n muy grande no nos sirve (en el juez sugerido las restricciones nos dicen que pueden llegar a vales 10^6 cada uno, lo que nos llevaría a una tabla de tamaño 10^{12}!!! algo inmanejable). Pero esto tiene fácil solución, y es que si nos fijamos en el código, solo necesitamos las dos últimas filas de la tabla. Por tanto aplicamos lo visto para el problema de Fibonacci en la sección 2.1 para mejorar la eficiencia en términos de espacio (también es importante, no solo la de tiempo). Es decir, originalmente nuestra memoria usada era O(nx) y ahora será de O(x). Por tanto la solución final sería:

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef long double ld;

// MOD
const ll mod = 1e9 + 7;

const int NUM_FILAS = 2;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);

    // INPUT
    ll n,x;
    cin >> n >> x;

```

```
21      ll coins[n];
22      for(int i = 0; i < n; ++i){
23          cin >> coins[i];
24      }
25
26      // SOLVE
27      int dp[NUM_FILAS][x+1];
28
29      // Caso base k = 0 (x != 0)
30      for(int i = 1; i ⩽ x; ++i){
31          dp[0][i] = 0;
32      }
33
34      // Caso base x = 0
35      for(int k = 0; k < NUM_FILAS; ++k){
36          dp[k][0] = 1;
37      }
38
39      // Caso general
40      for(int k = 1; k ⩽ n; ++k){
41          for(int i = 1; i ⩽ x; ++i){
42              dp[k % NUM_FILAS][i] = dp[(k-1) % NUM_FILAS][i];
43              if(coins[k-1] ⩽ i){
44                  dp[k % NUM_FILAS][i] += dp[k % NUM_FILAS][i-coins[k
    -1]];
45                  dp[k % NUM_FILAS][i] %= mod;
46              }
47          }
48      }
49
50      // OUTPUT
51      cout << dp[n % NUM_FILAS][x] << endl;
52
53      return 0;
54  }
```

Ejercicio 2.5.2. Knapsack II

3. Grafos

3.1. Nociones básicas de grafos

Terminología de grafos

Un **grafo** es un conjunto de **nodos** y **arcos**. Tradicionalmente en Informática se les llama de esta manera, mientras que en Matemáticas se les suele llamar vértices a los nodos y aristas a los arcos.

Notación. Nosotros emplearemos la notación estándar en Informática de nodos y arcos. Además de ahora en adelante se entiende que n es el número de nodos del grafo y m el número de arcos.

Los nodos representan **entes**, como ciudades, personas, objetos..., *estados* como los de un juego (por ejemplo ganar, perder y empate), u *opciones* (ir arriba, abajo, derecha o izquierda en un mapa; borrar una palabra u otra; coger un elemento u otro...) entre los que existe alguna **relación** (entre ciudades puede ser si están conectadas por carreteras o no, entre personas si son amigas o no, mientras que entre estados sería si se puede pasar de un estado a otro) la cual es representada por arcos. Es decir, si dos *entes* (seáse nodos) están relacionados entre sí (mediante una relación previamente establecida), entonces existe un arco que los *conecta (o relaciona)*. Recíprocamente, si existe un arco *conectando* dos nodos, entonces esos dos nodos están relacionados.

Se suelen representar que hay un *arco* (o relación) entre dos nodos como un de par compuesto por ambos nodos, es decir, si existe un arco entre los nodos 0 y 3, tendríamos:

Nodos: 0,3

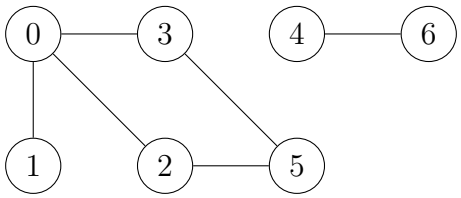
Arcos: (0,3)

Los grafos se representan gráficamente mediante puntos en un plano (nodos) y las líneas que unen dichos puntos (arcos). La siguiente página web es bastante útil a la hora de dibujarlos: Dibujar grafos

Ejemplo 3.1.1. Representación estándar de un grafo con 7 nodos y 6 arcos (arbitrarios).

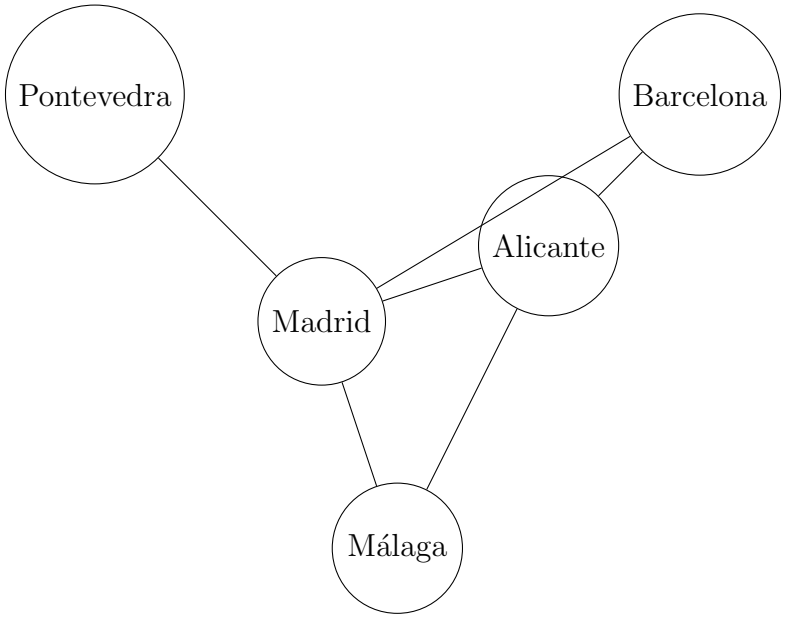
Nodos: 0,1,2,3,4,5,6

Arcos: (0,1), (0,2), (0,3), (3,5), (4,6), (2,5)



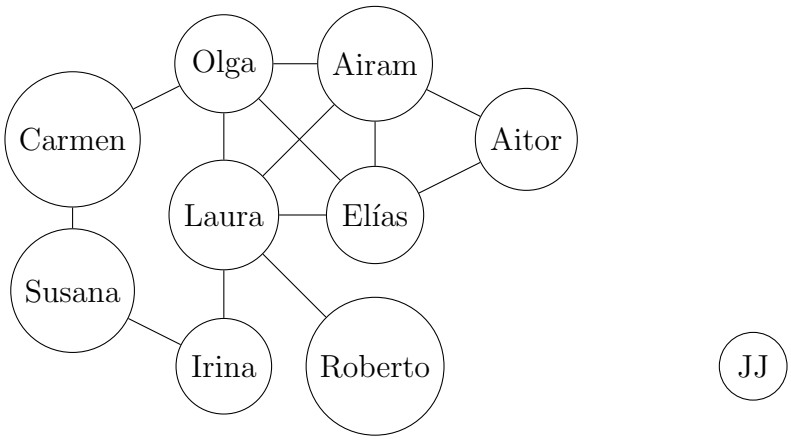
Los grafos nos permiten modelar problemas o situaciones de la vida real. Una analogía típica sería la identificar los nodos con ciudades y los arcos con las carreteras que las unen.

Ejemplo 3.1.2. Representemos un subconjunto de las carreteras que conectan distintas ciudades en España. Tenemos pues que Madrid está conectada con todas las ciudades a considerara por ser la capital. Pontevedra solo está conectada con Madrid porque al estar en Galicia está muy mal conectada. Por otro lado Alicante está conectada con Málaga y con Barcelona para que todos los guiris puedan recorrer las costa Mediterránea española sin tener que pasar por Madrid. El grafo que modela este sistema de carreteras se representaría de la siguiente forma:



Otro problema a modelar por grafos podría ser identificar los nodos con personas y la existencia de un arco entre dos nodos podría venir a representar que existe una relación de amistad entre ellas.

Ejemplo 3.1.3. Sean pues 10 personas: Laura, Elías, Airam, Olga, Irina, Susana, Carmen, Aitor, JJ (José Juan) y Roberto. Y sabemos que Laura, Elías, Airam, Olga son todos amigos entre sí; Irina es amiga de Laura y de Susana; Carmen es amiga de Susana y de Olga; Aitor es amigo de Elías y de Airam; Roberto es amigo de Laura y JJ no es amigo de nadie. Claramente las personas serán los nodos en nuestro problema y sus relaciones de amistad los arcos.



Una modelización muy útil también es la de representar un "mapa" (séase un tablero con diversas casillas) y los movimientos que se pueden realizar desde cada posición mediante un grafo.

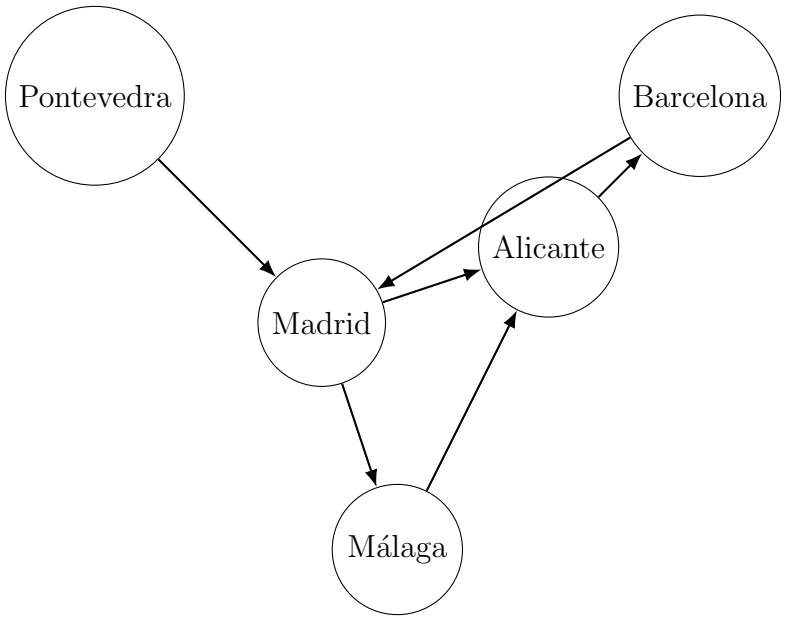
Ejemplo 3.1.4. Supongamos pues que tenemos el siguiente mapa donde X representa un obstáculo, una pared o simplemente una casilla a la que no te puedes mover.

Grafos dirigidos

Aunque no se mencionó antes, todos los grafos de ejemplo eran lo que se llaman grafos **no dirigidos o bidireccionales**. Estos grafos son aquellos en los que si existe un arco del nodo u al nodo v, entonces también existe del nodo v al nodo u. Los grafos que no cumplen está propiedad se llaman grafos **dirigidos o unidireccionales**.

En el ejemplo 3.1.2 de las carreteras y las provincias, tenemos un grafo no dirigido puesto que se asume que si se puede ir de Málaga a Madrid también se puede ir de Madrid a Málaga, es decir, se entiende que las carreteras son de doble sentido.

Ejemplo 3.1.5. Consideremos ahora que en vez de modelar las carreteras entre las distintas provincias, consideramos los vuelos de avión que hay disponibles de una provincia a otra. De forma que si queremos ir de Málaga a Madrid, debemos ir primero a Alicante, luego a Barcelona y ya por fin a Madrid, puesto que no hay vuelo directo de Málaga a Madrid (a pesar de si haberlo de Madrid a Málaga).



Notación. Se dice que un nodo u es **hijo** de un nodo v si existe un arco que va de u a v. Recíprocamente se dice que v es el **padre** de u. Alternativamente para grafos no dirigidos se suele usar la teminología de que u y v son vecinos.

Grafos ponderados

Grafos completos o densos

Ciclos

Árboles

Grados del grafo

grado de arcos que entran y salen y esas cosas

Conexión

Representación de grafos

Lista de Adyadcencia

Matriz de Adyacencia

Matriz de Incidencia

3.2. Algoritmos de búsqueda

Los algoritmos más elementales de grafos son los de **búsqueda o recorrido de grafos**. Es fundamental para casi todos los problemas saber recorrer un grafo de forma eficiente (y en el orden necesario), además de ser los algoritmos de base para resolver numerosos problemas.

Nota. Dichos algoritmos "visitan" cada nodo una **única** vez. De otra forma, en el caso de un grafo cíclico el algoritmo podría no tener fin.

3.2.1. Búsqueda en profundidad (DFS)

El algoritmo de *búsqueda en profundidad o depth first search (dfs)* es problablemente el algoritmo más básico de grafos y el más utilizado. Consiste en recorrer los nodos de un grafo de la siguiente manera: al "visitar" un nodo, procede a visitar uno de sus hijos, y luego a uno de los hijos del hijo escogido y así sucesivamente. Notamos que el algoritmo para escoger el hijo a visitar por defecto es el orden en el que se encuentran en la *lista de adyacencia*. Intuitivamente se ve que este algoritmo se implementa recursivamente.
Estudiemos primero la búsqueda en profundidad en árboles, por ser más sencilla.

Ejemplo 3.2.1. Análicemos paso por paso el orden en el que el algoritmo *dfs* recorrería el siguiente *árbol*, empezando desde el nodo 0. Observense los diagramas de la siguiente página.

Implementación árboles

La implementación para árboles generalizando a los no dirigidos es la siguiente:

```
vector<vector<int>> adjacency_list; // Lista de adyacencia

void dfs(int node, int parent = -1){
    for(int neighbour : adjacency_list[node]){
        if(neighbour != parent)
            dfs(neighbour, node);
    }
}
```

Notamos que la condición para visitar un nodo de que no lo hayamos visitado previamente, en árboles se reduce a que el vecino en cuestión no sea el "padre" del nodo (entendemos en este contexto como padre al nodo que llamo a la función `dfs` sobre él), puesto que este ya lo hemos visitado.

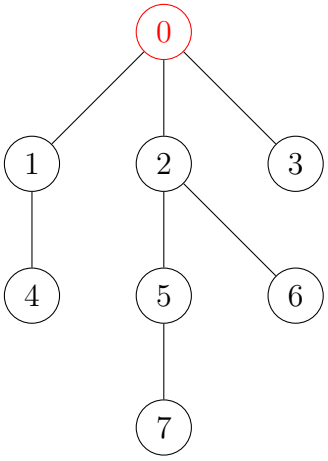
Observación. Notamos que en caso de tratarse de un árbol dirigido, esta condición no sería necesaria, dado que no podríamos ir de un nodo hijo a su padre.

Eficiencia implementación árboles

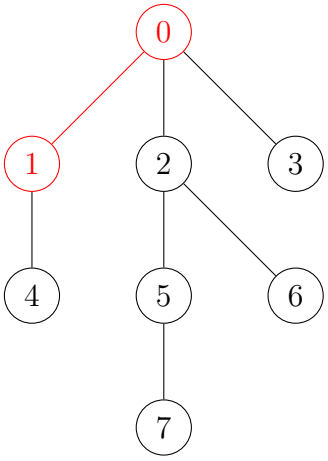
Análicemos la eficiencia teórica del algoritmo *dfs* para el caso de árboles. Su implementación consiste en una función recursiva, que tiene tantas llamadas recursivas como vecinos tenga el nodo en cuestión descontando al nodo padre.

Como se sabe que está función va a actuar sobre un árbol, se ve que no entrará en un ciclo infinito, es decir, aunque no haya un caso base claramente diferenciado, sabemos que en cuanto se llegue a un nodo hoja, es decir, un nodo cuyo único vecino sea su padre, no se realizarán más llamadas recursivas. Además tenemos que si n es el número de nodos, entonces el número de arcos es $n-1$, por tratarse de un árbol.

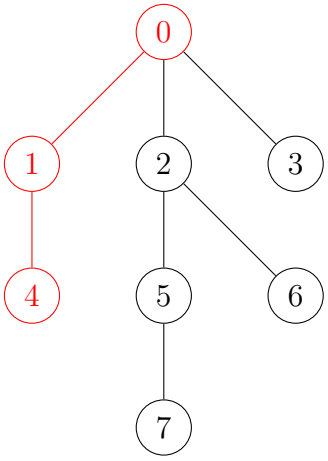
Es evidente pues, que la función se dedica a recorrer todos los nodos del árbol una **única** vez (por como están estructurados los árboles) y por tanto la eficiencia de la función es $O(n)$.
Comprobemos que esto es cierto de manera más formal.



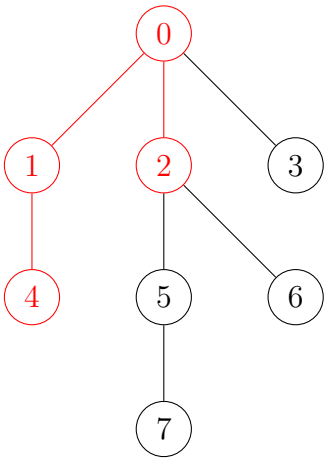
Visitamos el nodo 0



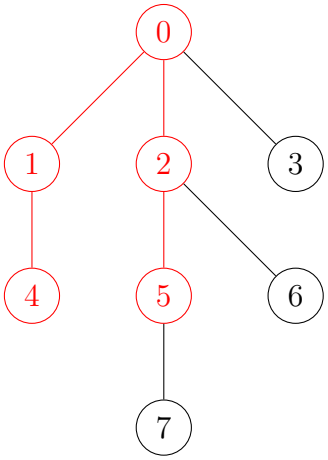
Visitamos 1 (hijo de 0)



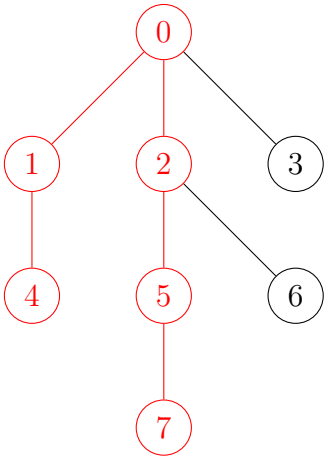
Visitamos 4 (hijo de 1)



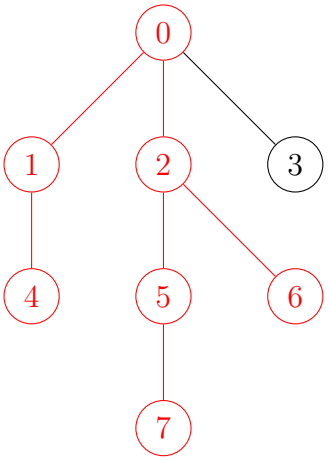
4 no tiene más hijos sin visitar, y 1 (padre de 4) tampoco por lo que visitamos 2 (que es el siguiente hijo de su padre 0)



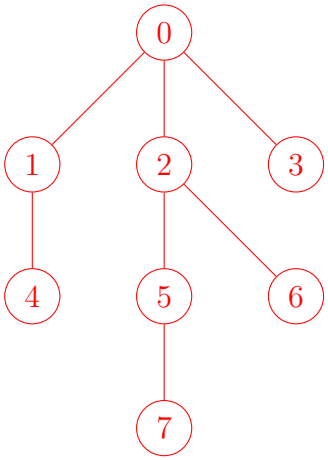
Visitamos 5 (hijo de 2)



Visitamos 7 (hijo de 5)



7 no tiene más hijos y 5 (padre de 7) tampoco, por lo que visitamos 6 (hijo de 2 (padre de 7))



6 no tiene más hijos y 2 (padre de 6) tampoco, por lo que visitamos 3 (hijo de 0 (padre de 2))

Tenemos pues que, llamando n al número de nodos del árbol, la función de eficiencia para la función `dfs` es:

$$T(n) = \sum_{i=1}^{nhijos} T(subarbol_i) + 1 \quad \forall n \in \mathbb{N}$$

donde $nhijos$ es el número de vecinos del nodo raíz y $subarbol_i$ es el número de nodos del subárbol del hijo i-ésimo.

Teorema 3.1. *Sea $T : \mathbb{N} \longrightarrow \mathbb{R}^+$ la función previamente definida. Entonces*

$$T(n) = n \quad \forall n \in \mathbb{N}$$

Demostración. Razonando por el **segundo principio de inducción**:

- Caso base $n = 1$:
  Obviamente $T(1) = 1$ puesto que al ser un árbol de un único nodo, este no tiene vecinos (y en consecuencia hijos) y por tanto la función no realiza llamadas recursivas.

- Supuesto cierto $T(k) = k \; \forall k < n$ tal que $k, n \in \mathbb{N}$ probemos que es cierto para n.
  Obviamente, si tenemos un árbol de n nodos, la suma de los subárboles de sus hijos debe ser $n-1$ ya que son todos los nodos del árbol inicial menos el nodo raíz, es decir: $\forall i \in \mathbb{N}$ tal que $1 \leqslant i \leqslant nhijos$

  $$\sum_{i=1}^{nhijos} subarbol_i = n - 1 \implies subarbol_i \leqslant n - 1 < n$$

  Por tanto, podemos aplicar la hipótesis de inducción a $subarbol_i$ $\forall i \in \mathbb{N}$ tal que $1 \leqslant i \leqslant nhijos$
  De esta forma:

  $$T(n) = \sum_{i=1}^{nhijos} T(subarbol_i)+1 = \{\text{hip. ind.}\} = \sum_{i=1}^{nhijos} subarbol_i+1 = n-1+1 = n$$

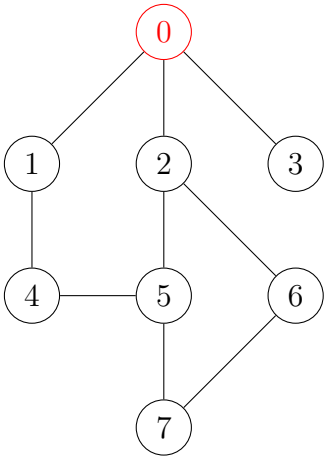
□

Queda pues demostrado que la función `dfs` es $O(n)$.

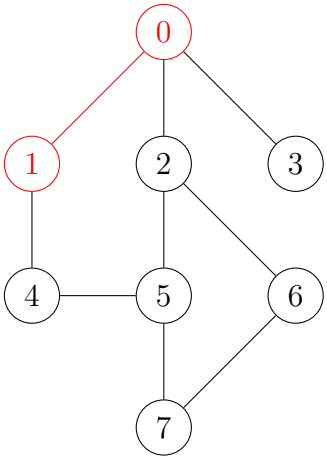
Pasamos ahora a una versión más general del algoritmo, válida para toda clase de grafos conexos. Análicemos primero como se comportaría el algoritmo con un grafo con ciclos con un ejemplo.

Ejemplo 3.2.2. Análicemos paso por paso el orden en el que el algoritmo *dfs* recorrería el siguiente grafo *cíclico*, empezando desde el nodo 0, que consiste en el árbol de ejemplo anterior pero añadiéndole unos cuantos arcos para formar ciclos. Obsérvense los diagramas de la siguiente página.

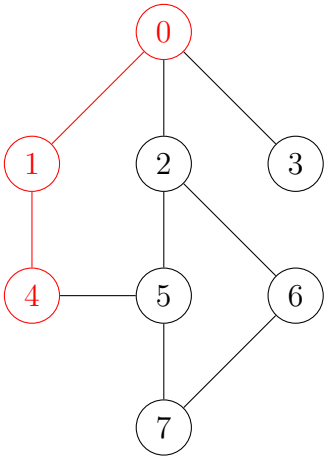
Cabe destacar como en este ejemplo hemos visitado mucho antes los nodos 5,7,6 antes que el nodo 2, a pesar de que 2 era hijo directo de 0, el nodo inicial y los otros nodos estaban más "abajo" en el grafo. Observamos además que en este caso aunque obviamente sí que hemos visitado todos los nodos, no hemos necesitado recorrer todos los arcos.



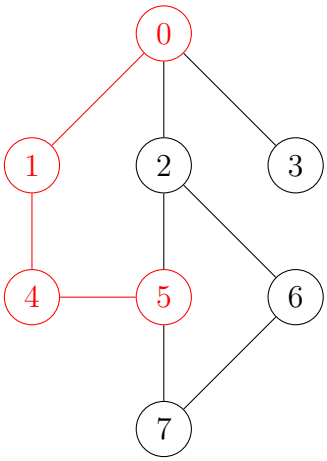
Visitamos el nodo 0



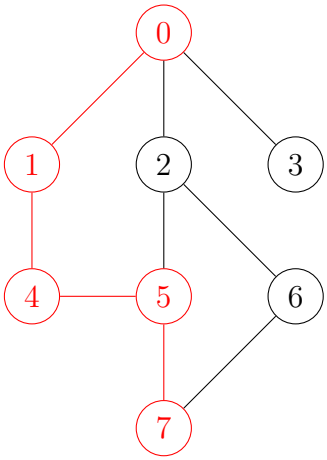
Visitamos 1 (hijo de 0)



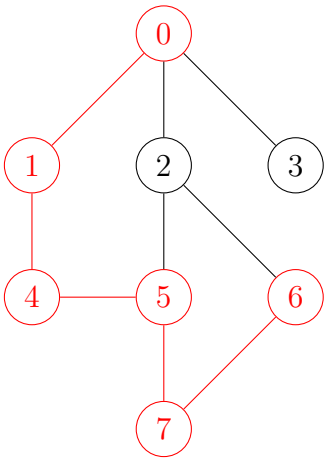
Visitamos 4 (hijo de 1)



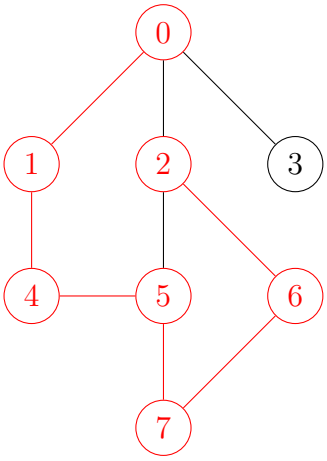
En este caso, 4 **sí** tiene más hijos, visitamos 5 (hijo de 4) **antes** de visitar 2



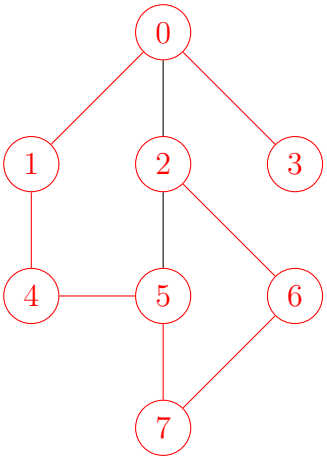
Visitamos 7 (hijo de 5) **antes** de visitar 2 también



En este caso 7 **sí** tiene más hijos, visitamos 6 (hijo de 7) **antes** de visitar 3



En este caso 6 **sí** tiene más hijos, visitamos 2 (como hijo de 6 y no de 0)



6 ya no tiene más hijos sin visitar, ni 7, ni 5, ni 4, ni 1. Visitamos 3 (hijo de 0)

Implementación general

Presentamos ahora una implementación válida para toda clase de grafos (representados por su correspondiente lista de adyacencia). En este caso para llevar la

cuenta de los nodos que ya han sido visitados necesitaremos un vector (o array) `visited` de `bool` inicializado a `false` que iremos actualizando a medida que vayamos recorriendo el grafo para evitar caer en bucles **infinitos**. De esta forma, el algoritmo a seguir es exactamente el mismo que para árboles, solo que cambiando la forma de comprobar si un nodo ha sido visitado o no.

```
vector<vector<int>> adjacency_list; // Lista de adyacencia

void dfs(int node, vector<bool> & visited){
    visited[node] = true;
    for(int neighbour : adjacency_list[node]){
        if(!visited[neighbour])
            dfs(neighbour, visited);
    }
}
```

Hay que tener en cuenta que el llamar a la función `dfs` sobre uno de los nodos del grafo solo hace que se recorra su **componente conexa**. Por tanto si estamos trabajando con un grafo que no es conexo y queremos recorrerlo en su totalidad, debemos llamar a la función `dfs` para cada componente conexa.

```
int n; // Numero de nodos del grafo
vector<bool> visited(n,false); // Vector para saber que
   nodos hemos visitado
for(int i = 0; i < n; ++i){
    if(!visited[i])
        dfs(i,visited);
}
```

De esta forma garantizamos que recorreremos el grafo al completo. Notamos que el vector `visited` no se resetea ni cambia del recorrido de una componente conexa a otra, y de esta forma evitamos recorrer la misma componente conexa más de una vez (en particular tantas veces como nodos tenga la componente).

Eficiencia versión general

En el caso general, tenemos que la eficiencia del algoritmo es $O(n+m)$.

Problemas

Veamos ahora este algoritmo aplicado a la resolución de diversos problemas. Se aconseja intentar algunos de estos problemas antes de mirar la solución. En todos los casos se ha facilitado el juez correspondiente para que se pueda enviar la solución y comprobar si esta correcta o no. Recordamos las soluciones de las problemas de programación **nunca** son únicas (aunque se pueden parecer) y no debe preocuparle que su solución no se parezca a la sugerida siempre y cuando el juez la acepte, es decir, el veredicto sea Accepted (AC).

Ejercicio 3.2.1. Counting Rooms

Enunciado: Tenemos el mapa de un edificio rectangular de tamaño $n \times m$ dividido en casillas del mismo tamaño. Cada casilla es suelo o pared. Podemos movernos arriba, abajo, izquierda y derecha (no podemos atravesar paredes). Nuestro objetivo es contar el número de habitaciones del edificio.

Solución:

Este problema se puede modelar mediante un grafo. Los nodos corresponderían a las casillas en las que podemos estar (suelo) del mapa. Los arcos representarían que podemos movernos de una casilla a otra, por tanto, tenemos que un nodo está conectado con todas sus casillas adyadcentes (sin contar diagonales).

Nos piden calcular el número de habitaciones del edificio. Tras haber modela el problema como un grafo, se ve claramente que está es una forma de pedirnos el número de componentes conexas del grafo, dado que una habitación consiste en un conjunto de casillas conectadas entre sí (se puede ir de cualquiera de ellas a cualquier otra).

Para calcular el número de componentes conexas de un grafo, nos valdremos de la última implementación de **DFS** que hemos visto. Estudiamos antes que dado un nodo el algoritmo `dfs` recorre solo la **componente conexa** a la que pertenece, por lo que haremos uso de el para "tachar" el resto de nodos de la componente cuando hayemos un nodo no visitado. De esta forma solo contabilizaremos un nodo por componente y tendremos lo deseado. Es decir, vamos iterando por los nodos no visitados del grafo. Cada vez que encontramos uno, contabilizamos su "habitación" y visitamos todos los nodos de su componente conexa para evitar contar la misma habitación dos veces.

Implementación:

```cpp
//https://cses.fi/problemset/task/1192
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef long double ld;

// Maximo valor que pueden tomar n y m acorde a las
// restricciones del problema
const int MAXN = 1e3;

int n,m; //Dimensiones del mapa
char grid[MAXN][MAXN]; // Mapa
bool visited[MAXN][MAXN]; // Saber si una casilla ha sido visitada
    o no

// Para moverse comodamente arriba, abajo, derecha e izquierda
const int sx[4] = {1,-1,0,0}; //eje x
const int sy[4] = {0,0,1,-1}; //eje y

// Comprobar si las coordenadas se encuentran dentro del mapa
bool validSquare(int x, int y){
    return 0 <= x && x < n && (0 <= y) && y < m;
}

// Comprobar si se puede ir a esas coordenadas
bool walkableSquare(int x, int y){
    return validSquare(x,y) && grid[x][y] == '.';
}

// Recorrer la componente conexa
void dfs(int x, int y){
    visited[x][y] = true;
```

```
34      for(int i = 0; i < 4; ++i){
35          int nx = x+sx[i];
36          int ny = y+sy[i];
37
38          if(walkableSquare(nx,ny) && !visited[nx][ny])
39              dfs(nx,ny);
40      }
41  }
42
43  int main(){
44      ios::sync_with_stdio(false);
45      cin.tie(0);
46
47      // INPUT
48      cin >> n >> m;
49
50      for(int i = 0; i < n; ++i){
51          for (int j = 0; j < m; j++){
52              cin >> grid[i][j];
53          }
54      }
55
56      // SOLVE
57      int sol = 0; // Numero de habitaciones (sease componentes
    conexas)
58      memset(visited,false,sizeof(visited));
59      // Buscamos contar el numero de componentes conexas
60      for(int i = 0; i < n; ++i){
61          for (int j = 0; j < m; j++){
62              // Si se puede ir a esa casilla y todavia no ha sido
    visitada
63              if(walkableSquare(i,j) && !visited[i][j]){
64                  ++sol; // Hemos encontrado una nueva componente
    conexa
65                  dfs(i,j); // La visitamos para evitar contarla mas
    de una vez
66              }
67          }
68      }
69
70      // OUTPUT
71      cout << sol << endl;
72
73      return 0;
74  }
```

3.2.2. Búsqueda en anchura (BFS)

3.3. Caminos mínimos

3.3.1. Algoritmo de Dijkstra

3.3.2. Algoritmo de Floyd-Warshall

3.4. Árbol Recubridor Mínimo (MST)

3.4.1. Estructura Union-find

3.4.2. Algoritmo de Kruskal

3.4.3. Algoritmo de Prim