La magia de la programación competitiva

Comunidad newliberty

25 de octubre de 2017

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Recursividad

1.1. Descripción y Motivación

Existen problemas que para resolverlos tenemos que ejecutar el mismo bloque de instrucciones varias veces, esto se puede lograr con ciclos iterativos o con recursividad. Todos los algoritmos iterativos pueden ser programados recursivamente y viceversa, aun que debemos aprender a elegir cual es la técnica correcta a utilizar. La implementación de un algoritmo iterativo consiste en repetir el cuerpo del bucle en cambio la implementacion de un algoritmo recursivo se basa en ejecutar repetidamente el mismo metodo. Los principales criterios a la hora de elegir entre programar algo iterativamente o recursivamente son: el rendimiento y la simpleza del codigo generado. Supongamos que debemos resolver el problema de sumar los primeros n numeros, dos algoritmos que solucionan este problema son los siguientes:

Iterativo

Listing 1.1: sumaIterativa.cpp

```
int sumaIterativa(int n){
    int resultado = 0;
    for(int i=1;i<=n;i++){
        resultado += i;
    }
    return resultado;
}
```

Recursivo

Listing 1.2: sumaRecursiva.cpp

```
int sumaRecursiva(int n){
    //Caso base
    if(n==1){
        return 1;
    }else{
        return n + sumaRecursiva(n-1);
    }
}
```

Algo muy importante a tener en cuenta en los algoritmos recursivos es el caso base, al igual que en los algoritmos iterativos se debe saber cuando detener la ejecución en los algoritmos recursivos necesitamos saber en donde detenernos. En realidad los dos algoritmos que mostramos tienen una ligera diferencia aun que dan el mismo resultado. En nuestro algoritmo iterativo sumamos desde 0 hasta n de la siguiente manera: $0 + 1 + 2 + 3 + ... + n$, pero en el recursivo sumamos desde n hasta 0: $n + (n - 1) + (n - 2)... + 0$. Si quisieramos que tuvieran un comportamiento mas similar podriamos programar el algoritmo recursivo de la siguiente manera:

Listing 1.3: sumaRecursiva2.cpp

```
int sumaRecursiva(int actual, int n){
    //Caso base
    if(actual==n){
        return actual;
    }else{
        return actual + sumaRecursiva(actual+1,n);
    }
}
```

El caso base esta muy ligado a la manera en que hacemos la recursividad, por lo general la recursividad se hace disminuyendo los parametros del problema, pero no siempre es asi como vimos en el segundo ejemplo, al igual que podemos hacer algoritmos iterativos con el contador ascendente o descendente y tenemos que generar la condicion de detener en base a este, en la recursividad también lo hacemos asi.
Quizas el ejemplo mas claro de recursividad es factorial de n. Ya que la solucion de factorial de n es $n * factorialde(n-1)$, y la solución de $factorialde(n-1) es (n-1) * factorialde(n-2)$ y asi consecutivamente. Por ejemplo factorial de 5 es:
$f(5) = 5 * f(4)$
$f(4) = 4 * f(3)$
$f(3) = 3 * f(2)$
$f(2) = 2 * f(1)$
$f(1) = 1$
por lo tanto
$f(2) = 2 * f(1) = 2 * 1 = 2$
$f(3) = 3 * f(2) = 3 * 2 = 6$
$f(4) = 4 * f(3) = 4 * 6 = 24$
$f(5) = 5 * f(4) = 5 * 12 = 120$
Mas o menos de esa manera funciona la recursividad en código, se van guardando cada llamada al metodo en una cola, al retornar regresa al metodo que la llamo. asi
$f(5) -> f(4) -> f(3) -> f(2) -> f(1)$
Iterativo

Listing 1.4: factorialIterativo.cpp

```
int factorial(int n){
    if(n==0)return 1;
    int resultado = 1;
    for(int i=n;i>=1;i--){
        resultado*=i;
    }
    return resultado;
}
```

Recursivo

Listing 1.5: factorialRecursivo.cpp

```
int factorial(int n){
    //Caso base
    if(n==0){
        return 1;
    }
    if(n==1){
        return 1;
    }
    return n * factorial(n-1);
}
```

Se debe tener cuidado al usar recursividad en no calcular muchas veces la misma solución, por ejemplo con el algoritmo de fibbonacci. su formula recursiva es $f(n) = f(n-1) + f(n-2)$. Si ejecutamos por ejemplo $f(5)$ sucederia lo siguiente:

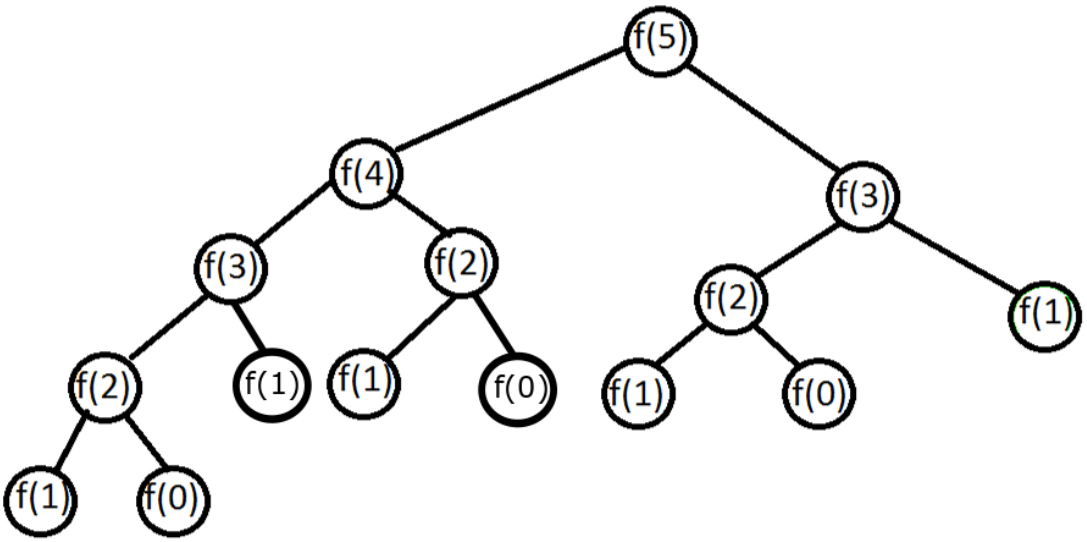


Figura 1.1: fibonacci.png

Podemos observar que recalculamos mucho, esto se puede resolver aplicando tecnicas de DP, pero eso lo veremos en otro capitulo.

1.2. Ejemplos

Ya vimos uno de los ejemplos mas tipicos de recursividad, el del factorial, en esta sección veremos el algoritmo de fibonacci, el algoritmo de Euclides (para hallar el máximo común divisor) y un algoritmo para solucionar las torres de hanoi.

Empecemos por el algoritmo de fibonacci, por definición la suseción de fibonacci comienza de la siguiente forma : 0,1,1,2,3,5,8 ... , cada elemento es la suma de sus dos anteriores. Más formalmente:

$f(n) = f(n-1) + f(n-2)$

Veamos primero como seria el algoritmo de fibonacci sin hacer uso de la recursión.

Listing 1.6: fibonacciIterativo.cpp

```
int fibonacci(int n){
    if(n==0)return 0;
    if(n==1)return 1;
    int a = 0;
    int b = 1;
    int c = a+b;
    for(int i=2;i<=n;i++){
        c = a+b;
        a = b;
        b = c;
    }
    return c;
}
```

Y ahora como seria usando recursión

Listing 1.7: fibonacciRecursivo.cpp

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

Mucho más simple, ¿no lo creen?. Ahora veamos el algoritmo de euclides
Iterativo

Listing 1.8: euclidesIterativo.cpp

```
int euclides(int a,int b){
    int temporal = a;
    while(a>0){
        temporal = a;
        a = b%a;
        b = temporal;
    }
    return b;
}
```

Recursivo

Listing 1.9: euclidesRecursivo.cpp

```
int euclides(int a,int b){
    if(b==0)return a;
    return euclides(b,a%b);
}
```

Por último mi ejemplo favorito para demostrar el potencial de la recursividad,las torres de hannoi. Si no conoces este juego, te recomiendo que primero busques en google “torres de hannoi online”, te saldran multiples opciones para jugarlo, es bastante simple e interesante.

En este caso no pondre una solución iterativa puesto que no se me ocurre ninguna, excepto simulando el comportamiento de la recursividad con una cola, (para saber más detalles al respecto, te invito a profundizar en como funciona internamente la recursividad).
El caso base de esta solución consiste en tener unicamente dos piezas apiladas, saber donde estan apiladas, hacia donde se dirigen, y el otro palo sera nuestro auxiliar.
La solución al caso base es muy sencilla, unicamente debemos desplazar la ficha superior a nuestro palo auxiliar, la ficha base a nuestro palo destino y por ultimo la ficha superior a nuestro destino, y

asi logramos resolver la torre de hannoi de nuestro caso base.

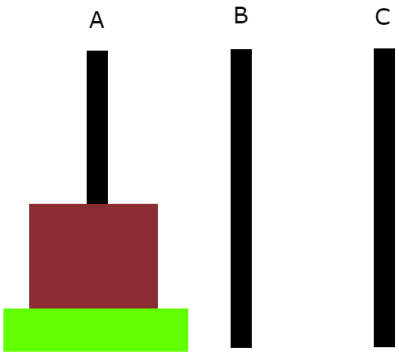


Figura 1.2: torre1.png

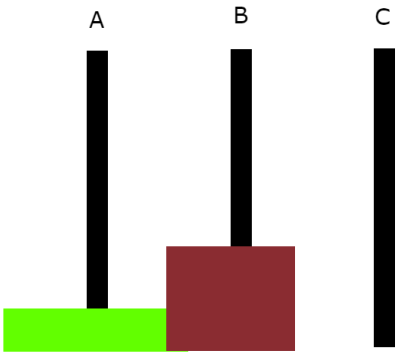


Figura 1.3: torre2.png

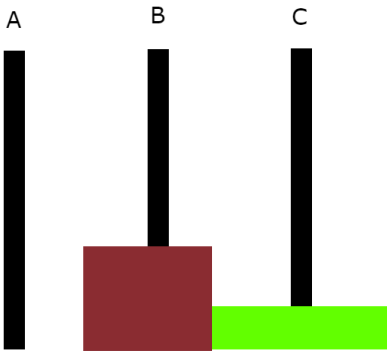


Figura 1.4: torre3.png

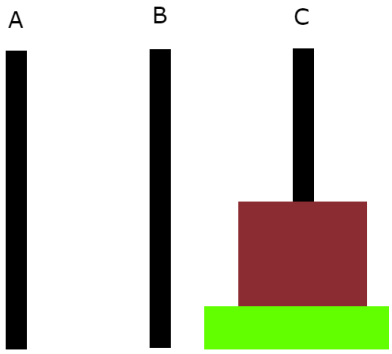


Figura 1.5: torre4.png

Pero que pasaria si fueran mas de dos fichas, he aqui donde viene la recursividad sucederia algo asi

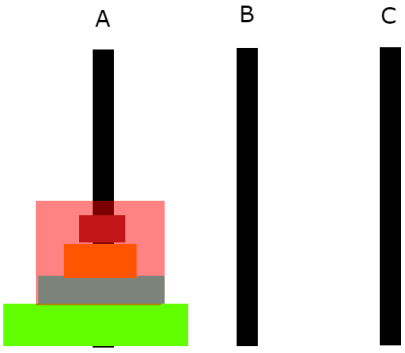


Figura 1.6: torre1-2.png

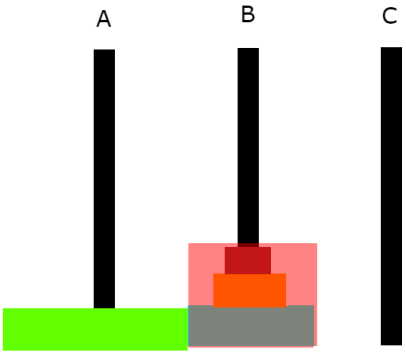


Figura 1.7: torre2-2.png

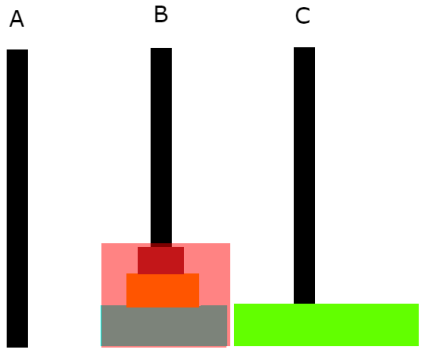


Figura 1.8: torre3-2.png

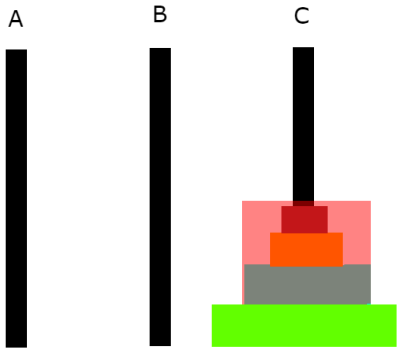


Figura 1.9: torre4-2.png

Internamente la recursion de las fichas sombreadas en rojo desde "torre1-2.png" hacia "torre2-2.png" funcionarian de la siguiente manera:

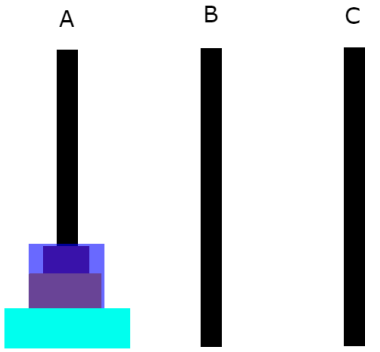


Figura 1.10: torre1-3.png

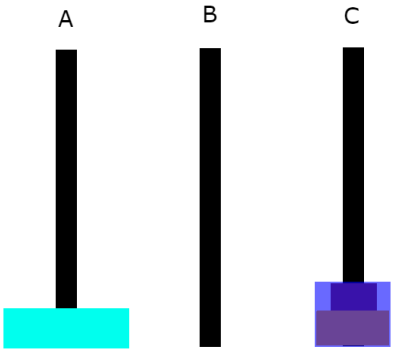


Figura 1.11: torre2-3.png

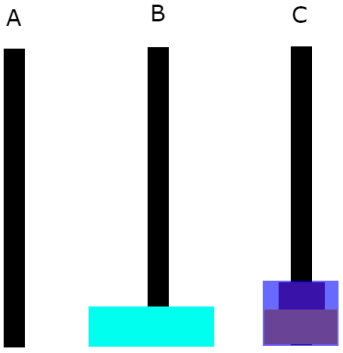


Figura 1.12: torre3-3.png

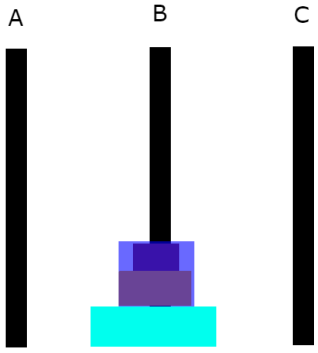


Figura 1.13: torre4-3.png

Y asi sucesivamente (recursivamente). La solución recursiva de la torre de hannoi consiste en llevar la parte superior (todas las piezas menos la base) hacia el palo auxiliar, mover la base al palo destino y finalmente mover la parte superior al palo destino. Cuando la parte superior es de mas de una pieza, se realiza la recursión cambiando invirtiendo el palo destino y el auxiliar. En código seria asi:

Recursivo

Listing 1.10: hannoi.cpp

```
1  void Hanoi(int disco, char origen, char intermedio, char destino){
2
3    if(disco == 1){
4      //caso base, solo movemos el disco a su destino
5      cout << "Mover disco " << disco << " desde " << origen << " hasta " <<
           destino << endl;
6    }else{
7      //movemos la parte superior al intermedio
8      Hanoi(disco-1, origen,destino,intermedio);
9      cout << "Mover disco " << disco << " desde " << origen << " hasta " <<
           destino << endl;
10     //movemos la parte superior al destino
11     Hanoi(disco-1,intermedio,origen,destino);
12   }
13 }
14
15 int main(){
16   int discos;
17   cout << "Ingrese la cantidad de discos: " << endl;
18   cin >> discos;
19   Hanoi(discos, 'A', 'B', 'C');
20
21   system("pause");
22 }
```

Capítulo 2

Matemáticas

2.1. Sucesiones y series

2.1.1. Sucesión aritmética

Las sucesiones aritméticas son aquellas que restando un elemento con su antecesor siempre da una constante se representan de la siguiente manera.
$an + b$
donde a es la resta entre dos elementos consecutivos y b es el primer elemento

2.1.2. Sucesión geométrica

Las sucesiones geométricas son aquellas que el cociente de un elemento con su antecesor siempre da una constante se representan de la siguiente manera.
ar^{n-1}
donde a es el primer termino y r es el cociente entre un numero y su anterior

2.1.3. Serie aritmética

Una serie aritmética es una sucesión creada con la suma de los términos de una sucesión aritmética, su formula es:
$a\frac{n(n+1)}{2} + nb$

2.1.4. Serie geométrica

Una serie geométrica es una sucesión creada con la suma del los términos de una sucesión geométrica, su formula es:
$a\frac{1-r^n}{1-r}$

2.2. Sumatorios

Los sumatorios son la suma de elementos de una secuencia, estas son las propiedades:

- la cantidad de elementos de un sumatorio es el limite superior menos el limite inferior mas la unidad
- el sumatorio de una constante es la cantidad de elementos por la constante
- el sumatorio es una transformación lineal o aplicación lineal y cumple con todas sus propiedades

2.3. Teoria de numeros

2.3.1. Máximo común divisor y minino común múltiplo

Máximo común divisor

El máximo común divisor se cal calcula con el algoritmo de Euclides si $b = 0$ $gcd(a, b) = a$
de lo contrario $gcd(a, b) = gcd(b, r)$ donde $r = a \mod b$
c++ ya tiene implementada esta función en su algorithm como __gcd

mínimo común múltiplo

$lcd(a, b) = \frac{ab}{gcd(a,b)}$

2.3.2. Algoritmo de Euclides extendido

El algoritmo de Euclides extendido sirve para hallar 2 números t y s que dados a y b
$at + bs = gcd(a, b)$

Listing 2.1: euclidesExtendido.cpp

```
int x,y,d;
void euclidesExtendido(int a, int b) {
    //caso base
        if (b == 0) {
        x = 1;
        y = 0;
        d = a;
        return;
    }
        euclidesExtendido(b, a % b);
        int x1 = y;
        int y1 = x - (a / b) * y;
        x = x1;
        y = y1;
}
```

2.3.3. Ecuaciones diofanticas lineales de 2 variables

Estas ecuaciones son de la forma $ax + by = c$ donde a, b y c son números enteros y el problema se encuentra en calcular 2 enteros x e y que satisfagan la ecuación, la ecuación tiene múltiples soluciones y sirven comúnmente para solucionar congruencias como $ax+b \equiv cx+d \mod m$ siendo a,b,c,d números conocidos utilizando las propiedades de la aritmética modular tenemos
$(ax + b) - (cx + d) = ym$
$(a - c)x + (b - d) = ym$
$(c - a)x + ym = (b - d)$
para solucionar la ecuación de la forma $ax + by = c$ tenemos que garantizar que $gcd(a, b)|c$ si no se cumple esta condición la ecuación no tendrá soluciones enteras si se cumple utilizamos el algoritmo de Euclides y hallamos s y t $at + bs = gcd(a, b)$ multiplicamos por c a ambos lados y dividimos por $gcd(a, b)$ y así hallamos nuestra primera solución donde $x_0 = \frac{tc}{gcd(a,b)}$ y $y_0 = \frac{st}{gcd(a,b)}$
las siguientes soluciones son de la forma $x = x_0 + \frac{b}{d}n$ y $y = y_0 - \frac{a}{d}n$

2.3.4. conjunto z_n

El conjunto z_n es el conjunto de elementos $[0, 1, 2, ...n - 1]$

2.3.5. Propiedades de la aritmética modular

- si $a \equiv b \mod n$ entonces $a + c \equiv b + c \mod n$
- si $a \equiv b \mod n$ entonces $(a - b)|m$
- $a + b \mod n = (a \mod n + b \mod n) \mod n$
- $ab \mod n = ((a \mod n)(b \mod n)) \mod n$
- si $a \equiv b \mod n$ entonces $ac \equiv bc \mod n$

2.3.6. Inverso multiplicativo en z_n

El inverso multiplicativo de un numero a en la aritmética modular en el conjunto z_n es encontrar un numero x que satisfaga $ax \equiv 1 \mod n$ para ello se usa el algoritmo de Euclides extendido que presentamos anteriormente.
para que a sea invertible a y n tienen que ser coprimos, a continuación se muestra el algoritmo para hallar el inverso

```
long inverse_Zn(int a,int n){
  extendedEuclid(a,n);
  if(d!=1){
        return -1;
  }
  else{
        if(x<0){
                x+=n;
                return x;
        }
  }
}
```

2.3.7. Números primos

Los números primos son todos aquellos que son divisibles unicamente por si mismos y por uno en el conjunto del los números naturales.

Criba de Eratostenes

Es un algoritmo para hallar todos los números primos desde 1 hasta un numero n y consiste en hacer una cuadricula con con los números y coger el primer primo elevarlo al cuadrado y tachar todos los números que a partir de su cuadrado sean múltiplos de este coger el siguiente numero primo elevarlo al cuadrado y repetir el proceso cuando el numero se pase de n terminamos y los números primos son lo que no hemos tachado.
Hallemos los números primos hasta el 100
tachamos el 1 y iniciamos con el 2 2x2=4 y a partir de ahí tachamos todos los múltiplos de 2

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Figura 2.1: criba-multiplos-de-2.png

ahora el siguiente número sin tachar es el 3, 3x3=9 y a partir de ahí todos los múltiplos de 3

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Figura 2.2: criba-multiplos-de-2y3.png

el siguiente numero sin tachar es el 5, 5x5=25 y a partir de ahí tachamos todos los múltiplos de 5,

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Figura 2.3: criba-multiplos-de-2-3y5.png

el siguiente 7, 7x7=49 y a partir de ahí tachamos todos los múltiplos del 7

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Figura 2.4: criba-multiplos-de-2-3-5y7.png

y terminamos por que el siguiente numero es 11 y 11x11= 121 que se pasa de nuestro rango, los números primos son los que no han sido marcados.

Listing 2.2: criba.cpp

```
const int MAXN = 100;
bool criba[MAXN + 5];
vector<int> primos;
void construir_criba()
{
    memset(criba, false, sizeof(criba));
    criba[0] = criba[1] = true;
    for (int i = 2; i * i <= MAXN; i++) {
        //Coger el proximo que no este marcado
        if (!criba[i]) {
            for (int j = i * i; j <= MAXN; j += i) {
                //Marcar sus multiplos
                criba[j] = true;
            }
        }
    }
    for (int i = 2; i <= MAXN; ++i) {
        if (!criba[i])
            primos.push_back(i);
    }
}
```

Este código fue sacado y editado de las presentaciones de la universidad EAFIT ver [1]

Comprobar si un número es primo

Para saber si un un numero p es primo, si p es menor que nuestro n retornamos la negación de la criba, si no comprobamos si este es divisible por alguno de los primos que tenemos, si alguno lo divide significa que no es primo, esto solo funciona si p es menor que el ultimo primo que tengamos al cuadrado.

Listing 2.3: criba.cpp

```
bool esPrimo(long long N)
{
    if (N < criba.size())
        return !criba[N];

    for (int i = 0; i < (int)primos.size(); i++)
        if (N % primos[i] == 0)
            retu\begin{lstlisting}[style=C,caption=criba.cpp]

rn false;
    return true;
}
```

2.3.8. Floyd's Cycle-Finding

Es una técnica para detectar un ciclo dentro de una secuencia creada con un función $f(x)$de la forma $g(x) \mod M$ donde el primer valor se da y el resto de elementos se generaría de esta manera $\{x_0, x_1 = f(x_0), x_2 = f(x_1), \dots\}$, si tenemos la función $f(x) = 3x + 5 \mod 12$ con $x_0 = 3$ tendríamos $3, 2, 11, 2, 11, 2$ Nuestro objetivo es encontrar 2 valores
mu =cantidad de numeros que hay antes de que inicie el ciclo
$lambda$ =cantidad de elementos que tiene el ciclo
En este caso mu es 1 y $lambda$ es 2
El algoritmo de detección de ciclo se hace con la analogía de la liebre y la tortuga y tiene 3 pasos

primer paso:

iniciamos la tortuga en $f(x_0)$ y la libre en $(f(x_0))$ la libre se mueve 2 veces avanzamos la tortuga $f(tortuga)$ y la libre $f(f(liebre))$ hasta que los 2 punteros coincidan

paso 2:

iniciamos $mu = 0$ hacemos la liebre igual a nuestro inicio y empezamos iterar los 2 punteros paso a paso sumando le 1 a mu hasta que coincidan.

paso 3:

estando los dos punteros en el mismo lugar iniciamos $lambda = 1$ y $libre = f(liebre)$ e iteramos solo con la liebre hasta que vuelva a coincidir con la tortuga sumándole 1 a $lambda$ por cada iteración.

Listing 2.4: floydCycleFinding.cpp

```
ii floydCycleFinding(int x0)
{
    // paso 1:
    int tortuga = f(x0), liebre = f(f(x0));
    while (tortuga != liebre) {
        tortuga = f(tortuga);
        liebre = f(f(liebre));
    }
    // paso 2:
    int mu = 0;
    liebre = x0;
    while (tortuga != liebre) {
        tortuga = f(tortuga);
        liebre = f(liebre);
        mu++;
    }
    // paso 3:
    int lambda = 1;
    liebre = f(tortuga);
    while (tortuga != liebre) {
        liebre = f(liebre);
        lambda++;
    }
    return ii(mu, lambda);
}
```

si quieres ver este problema de una forma mas visual puedes visitar [8]

2.4. Combinatoria

2.4.1. Principio multiplicativo

Si se quiere realizar un procedimiento de n pasos donde el primer paso puede ser hecho de a_1, el segundo paso de a_2 y así sucesivamente hasta a_n las formas de llevar a cabo el procedimiento son $a_1 * a_2 ... * a_n$

2.4.2. Número de permutaciones de n elementos

El número de permutaciones es el numero de arreglos donde el orden importa, el numero de permutaciones se calcula como $P(n, n) = n!$

2.4.3. Número de permutaciones de n elementos tomados de a m

El numero de permutaciones de n elementos tomados de a m son $P(n, m) = \frac{n!}{(n-m)!}$

2.4.4. Número de permutaciones de n elementos tomados de a m con repetición

En este problema tenemos un suministro ilimitado de los n elementos diferentes y queremos saber de cuantas maneras podemos coger m elementos. su formula es: $Pr(n, m) = n^m$

2.4.5. Número de permutaciones con al menos un elemento fijo

El número de permutaciones que tienen al menos un elemento fijo son todas las permutaciones que no son desarreglos
$n! - D_n$

2.4.6. Número de permutaciones donde el primer elemento se repite a veces el segundo b veces ...

el número de permutaciones es: $\frac{n!}{a!b!...}$

2.4.7. Número de desarreglos

El numero de desarreglos es el numero de permutaciones que podemos hacer donde ninguno de los elementos esta en su posición inicial, se calculan con la siguiente formula recursiva.
$D_n = (n-1)(D_{n-1} + D_{n-2})$
casos base: $D_2 = 1$ $D_3 = 2$

2.4.8. Número de permutaciones de n elementos que dejan exactamente k elementos fijos

El numero de permutaciones que dejan exactamente k elementos fijos es lo mismo que tachar k elementos y hacer un desarreglo con los n-k restantes. entonces la formula seria el numero de formas que podemos escoger k elementos del total multiplicado el desarreglo de n-k, siendo $s(n,k)$ el numero de arreglos con exactamente k elementos fijos tenemos:
$s(n,k) = c(n,k)D_{n-k}$

2.4.9. Número de combinaciones de n elementos tomados de a m

el numero de combinaciones de n elementos cogidos de m son el numero de formas que podemos coger m elementos de los n sin importar su orden

Formula:

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

Formula recursiva:

casos bases $\binom{m}{m} = \binom{m}{0} = 1$
de mas casos $\binom{m}{n} = \binom{m-1}{n} + \binom{m-1}{n-1}$

Propiedades de los números combinatorios:

1) $\binom{m}{n} = \binom{m}{m-n}$
2) $\binom{m}{m-1} = m$
3) $\binom{m}{1} = m$

2.4.10. Números figurados

los números figurados, son números enteros que son posibles representarlos como una figura geométrica, muchos de ellos tienen relación con la combinatoria

Números triangulares

Estos se pueden representar como un triangulo equilátero

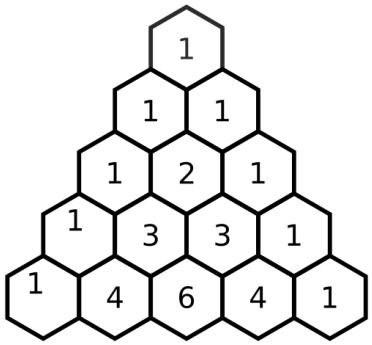


Figura 2.5: triangular.png

son la suma de los primeros n números naturales y su relación con la combinatoria es la siguiente:
Los números triangulares se encuentran en el triangulo de pascal en la tercera fila del triangulo de pascal

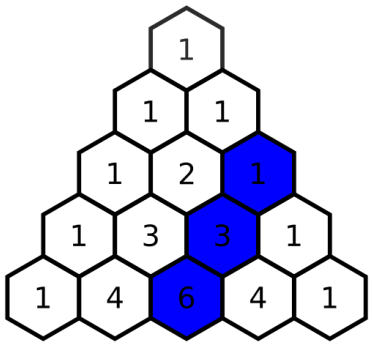


Figura 2.6: triangulares-pascal.png

y el triangulo de pascal lo podemos representar como números combinatorios de la siguiente forma:

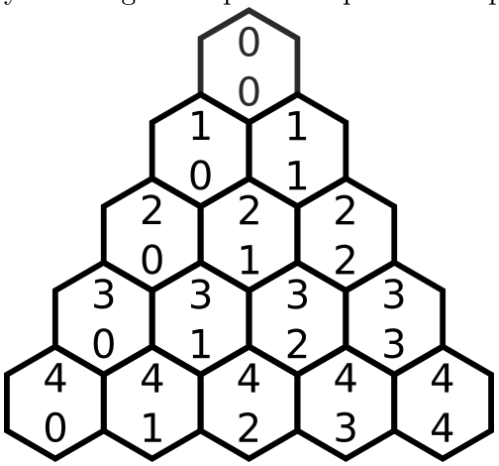


Figura 2.7: triangulo-de-pascal-combinatoria.png

Viendo en el triangulo de pascal podemos ver que podemos representar los números triangulares como la $T_n = \binom{n+1}{2}$ o usando las propiedades de los números combinados como $T_n = \binom{n+1}{n-1}$

Números cuadráticos

Los números cuadráticos se pueden representar como un cuadrado

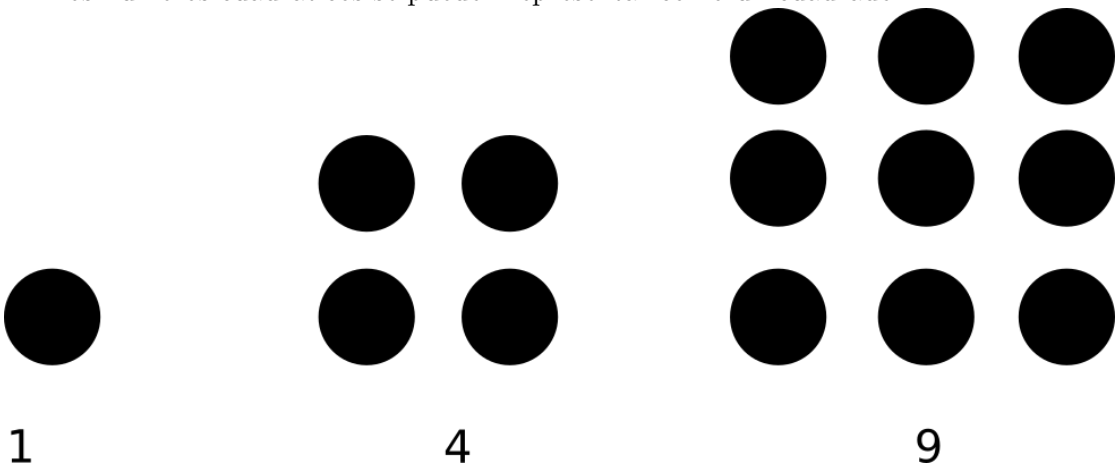


Figura 2.8: cuadraticos.png

tienen un propiedad algo extraña pero fascinante un numero cuadrático es la suma de dos números triangulares continuos así que $n^2 = T_n + T_{n-1}$

Números tetraédricos

Ahora pasamos a un espacio tridimensional, estos se representan en forma de tetraedro

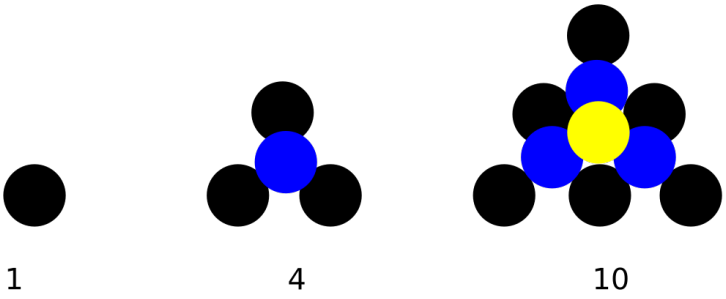


Figura 2.9: tetraedricos.png

un tetraedro es un poliedro de 4 caras triangulares, son la suma de lo primeros n números triangulares y la 4 fila de triangulo de pascal

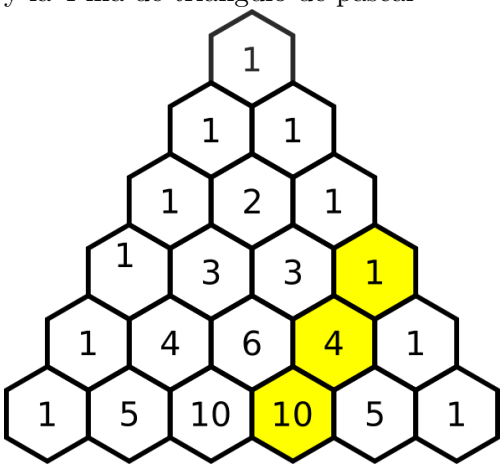


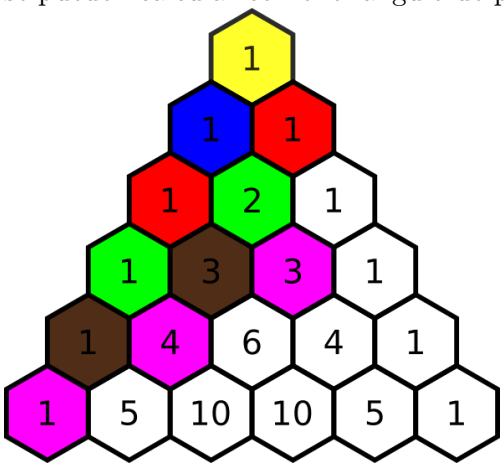
Figura 2.10: tetraedricos-pascal.png

en el triangulo de pascal podemos ver que podemos representar los números tetraédricos como la

$Tr_n = \binom{n+2}{3}$ o usando las propiedades de los números combinados como $Tr_n = \binom{n+2}{n-1}$

2.4.11. Números de fibonacci

Los números de fibonacci ademas de aparecer en muchos de los patrones de la naturaleza también se pueden calcular con el triangulo de pascal



$fib(n+1) = \sum_{k=0}^{\frac{n}{2}} \binom{n-k}{k}$

Figura 2.11: fibonacci-pascal.png

2.4.12. Números de catalán

Los números de catalán son una secuencia de números naturales definidos como $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!}$ con $n \geq 0$. como era de esperarse estos también pueden ser calculados con el triangulo de pascal

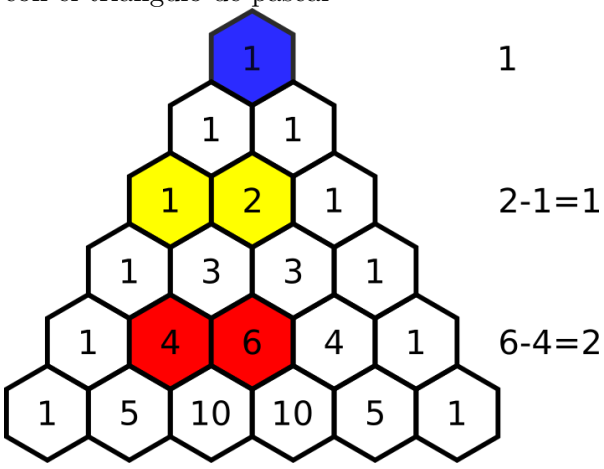


Figura 2.12: catalan-pascal.png

y su fórmula con números combinatorios es $C_n = \binom{2n}{n} - \binom{2n}{n-1}$ con $n \geq 1$.

Aplicaciones de los números de catalán

- son el número de expresiones que tienen n pares de paréntesis correctamente colocados, para n=3 tenemos ((())) ()(()) ()()() (())() (()())

- son el número de formas de partir un polígono convexo de n+2 lados en triángulos para n=2 tenemos

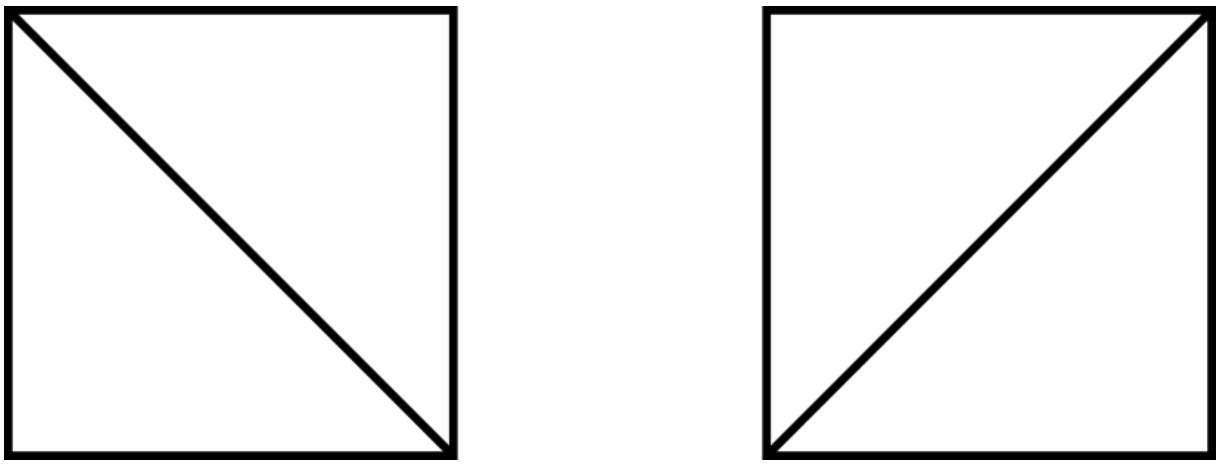


Figura 2.13: poligono-catalan.png

- número de árboles binarios que se pueden construir que tenga n+1 hojas en los que cada nodo tiene 0 ó 2 hijos, para n=2 tenemos

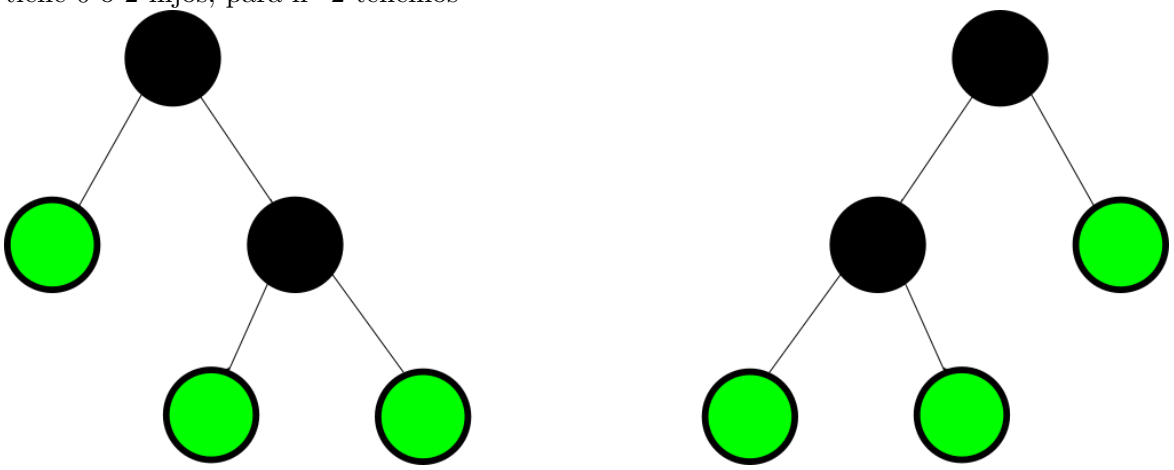


Figura 2.14: arbol-catalan.png

- número de caminos que se pueden trazar por las lineas de una cuadricula de n*n sin atravesar la diagonal, para n=2 tenemos

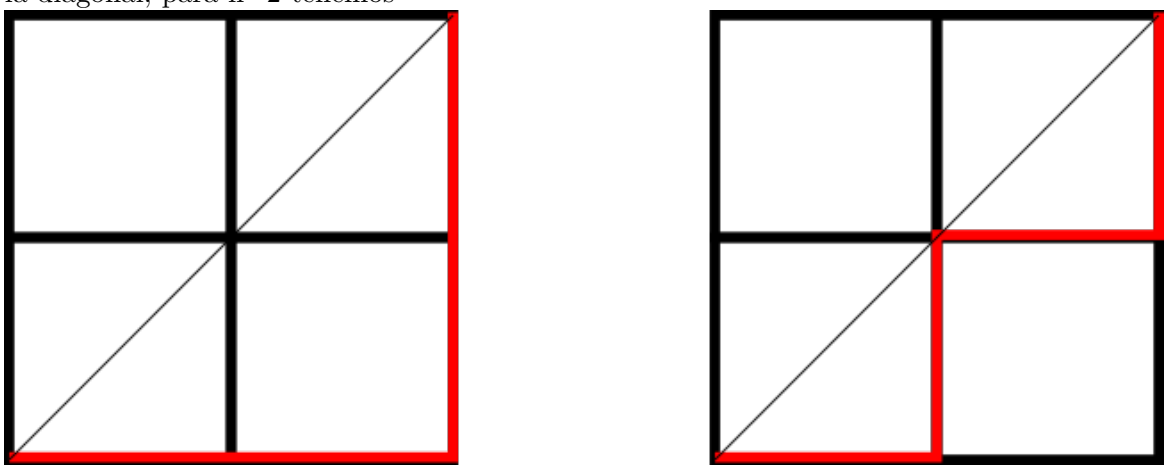


Figura 2.15: caminos-catalan.png

2.5. Probabilidad

2.5.1. Regla de Laplace

La regla de laplace establece que la probabilidad de que ocurra un evento es la cantidad de casos favorables sobre la cantidad de casos posibles

$p(x) = \frac{casos\_favorables}{casos\_posibles}$

2.5.2. Probabilidad de intersección de sucesos

Si tenemos dos sucesos a y b la probabilidad de que suceda a y b es: $p(a \cap b) = p(a)P(b|a)$

2.5.3. Probabilidad de unión de sucesos

Si tenemos dos sucesos a y b la probabilidad de que suceda a ó b es: $p(a \cup b) = p(a) + p(b) - p(a \cap b)$

2.5.4. Probabilidad condicionada

La probabilidad condicionada es la probabilidad de que ocurra un evento a sabiendo que ya ocurrió un evento b y se calcula de la siguiente manera $p(a|b) = \frac{p(a \cap b)}{p(b)}$

2.5.5. Teorema de Bayes

El teorema de Bayes indica una relación entre $p(a|b)$ y $p(b|a)$ y puede ser sacado de las formulas anteriores que hemos visto $p(a|b) = \frac{p(a)P(b|a)}{p(b)}$

2.6. Potenciación rápida

2.6.1. Introducción

La potenciación rápida es un algoritmo para calcular la potencia enésima de cualquier estructura donde este definida la multiplicación y el algoritmo es el siguiente:

mientras exponente sea diferente de 0 se repiten los siguientes pasos

- hacemos el resultado igual a la unidad, si el exponente es impar multiplicamos el resultado por nuestra base osea $resultado = resultado * base$
- hacemos la $base = base * base$
- tomamos la parte entera de dividir nuestro exponente por 2 $exponente = exponente/2$, estamos utilizando la propiedad de la potenciación que dice $(2^n)^m = 2^{mn}$

Ejemplo con un números enteros:

resultado	base	exponente
1	2	13
2	4	6
2	16	3
32	256	1
8192	65536	0

Cuadro 2.1: potenciacion rápida con números enteros.

$2^{13} = 2 * \left(2^2\right)^6$
$2^{13} = 2 * \left(2^4\right)^3$
$2^{13} = 2 * 2^4 \left(2^8\right)^1$
$2^{13} = 2 * 2^4 * 2^8 \left(2^{16}\right)^0$

Algoritmo general

Listing 2.5: operadorPotencia

```
Estructura operator^(const int& n) const
{
    Extructurra resultado(), base = *this;
    int exponente = n;
    while (exponente) {
        if (exponente & 1)//comprueba si exponente es impar
            resultado = resultado * base;
        exponente = exponente >> 1; //es lo mismo que exponente=exponente/2;
        base = base * base;
    }
    return resultado;
}
```

2.6.2. Números complejos

Una de las aplicaciones que tiene la potenciación rápida es la potenciación de números complejos

Listing 2.6: PotenciacionComplejos.cpp

```
struct Complejo{
    int a, b;
    Complejo(int _a, int _b) {
        a = _a, b = _b;
    }
    Complejo operator*(const Complejo& x) const {
        Complejo resultado(this->a*x.a - this->b*x.b, this->a*x.b + this->b*
            x.a);
        return resultado;
    }
    Complejo operator^(const int& n) const {
        Complejo resultado(1,0), base = *this;
        int exponente = n;
        while(exponente) {
            if(exponente&1)     resultado = resultado * base;
            exponente = exponente>>1, base = base * base;
        }
        return resultado;
    }
};
```

2.6.3. Matrices

Otra de las aplicaciones del algoritmo general de potenciación, es el poder elevar una matriz a un numero entero positivo, esta operación solo es posible si la matriz es una matriz cuadrada

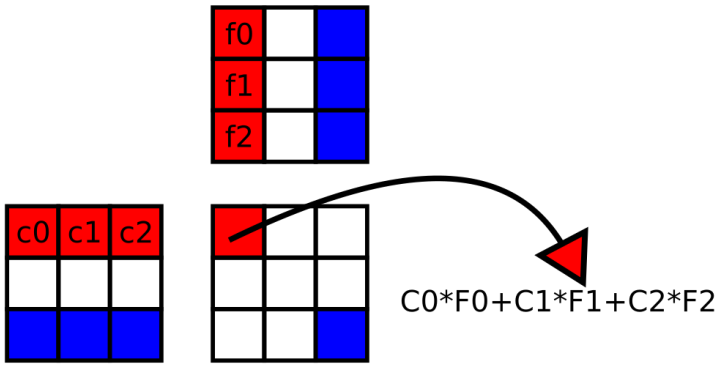


Figura 2.16: multiplicacion-de-matrices.png

En muchos de los ejercicios donde se aplica lo anterior nos piden que demos un resultado modulo algún numero es por eso que en el código se puede apreciar que sacamos el modulo después de hacer el producto punto entre una fila una columna.

Listing 2.7: PotenciacionMatrices.cpp

```
struct Matrix {
    int v[100][100];
    int row, col;
    Matrix(int n, int m, int a = 0) {
        memset(v, 0, sizeof(v));
        row = n, col = m;
        for(int i = 0; i < row && i < col; i++)
            v[i][i] = a;
    }
    Matrix operator*(const Matrix& x) const {
        Matrix resultado(row, x.col);
        for(int i = 0; i < row; i++) {
            for(int k = 0; k < col; k++) {
                if (v[i][k])
                for(int j = 0; j < x.col; j++) {
                    resultado.v[i][j] += v[i][k] * x.v[k][j],
                    resultado.v[i][j] %= mod;
                }
            }
        }
        return resultado;
    }
    Matrix operator^(const int& n) const {
        Matrix resultado(row, col, 1), base = *this;
        int exponente = n;
        while(exponente) {
            if(exponente&1)     resultado = resultado * base;
            exponente = exponente>>1, base = base * base;
        }
        return resultado;
    }
};
```

Cantidad de rutas que se pueden tomar con P pasajes

Una de las aplicaciones de la potenciación de matrices es encontrar la cantidad de rutas que puedo tomar en un grafo con P pasajes para llegar de un lugar a otro. Vamos a explicar el este problema con el siguiente grafo

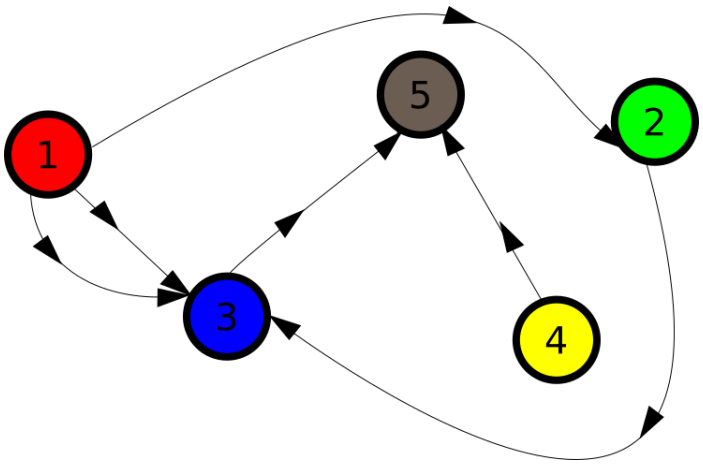


Figura 2.17: grafo-potenciacion.png

Creamos una matriz donde las filas son el nodo en el que me encuentro y la columnas el nodo al que quiero ir la intersección entre una fila y una columna es la cantidad de caminos directos que hay del nodo de la fila al nodo de la columna.

	1	2	3	4	5
1	0	1	2	0	0
2	0	0	1	0	0
3	0	0	0	0	1
4	0	0	0	0	1
5	0	0	0	0	0

Figura 2.18: matriz-de-grafo-potenciacion.png

Esta propiedad se basa en el principio multiplicativo que vimos anteriormente, si solo queremos saber de cuantas formas podemos ir de un nodo inicial I a un nodo destino D solo tenemos que consultar la matriz en la posición $[I][D]$ pero si queremos saber de cuantas formas podemos ir de I a D con 2 pasajes multiplicamos las posibilidades de ir de I a un nodo de transito y de ese nodo de transito a nuestro destino D y sumamos el resultado de todos los posibles nodos auxiliares esta es la misma operación que hacer un producto punto $fila * columna$, si en nuestro grafo queremos ir del nodo 1 al nodo 3 en 2 pasos solo tenemos que hacer lo siguiente: $I = 1$,$D = 3$, $matrix[1][1] * matrix[1][3] + matrix[1][2] * matrix[2][3] + matrix[1][3] * matrix[3][3] + matrix[1][4] * matrix[4][3] + matrix[1][5] * matrix[5][3]$ si queremos hallar la cantidad de formas que podemos ir de cualquier I a cualquier D tenemos que hacer la multiplicación de la matriz por ella misma y cada posición de la matriz tendrá la cantidad de formas que se puede llegar de cualquier nodo I a cualquier nodo D con 2 pasajes. Si queremos en hallar la cantidad de formas en que se puede ir de un nodo I a un nodo D en 3 pasajes elevamos la matriz a la 3 ya si sucesivamente dependiendo de lo que necesitemos.
Uno de los ejercicios que se resuelve de esta manera es teletransport que lo puedes ver en [7]

2.7. Transformaciones lineales

una transformación lineal es una función que satisface los siguientes axiomas

- $f(x + y) = f(x) + f(y)$
- $f(ax) = af(x)$ siendo a una constante

Capítulo 3

Geometricos

3.1. Estructuras geométricas

3.1.1. Puntos

Un punto es una estructura matemática que no tiene dimensión, solo describe una posición en el espacio. Pueden estar en el espacio 1d sobre una recta, 2d un plano ... nd. Sobre los puntos se pueden hacer varias operaciones que veremos mas adelante, la representación de un punto solo es un conjunto de coordenadas que describen su posición, para una dimensión tendríamos un numero x, para dos dimensiones 2 números x, y para 3 dimensiones x, y, z y para n dimensiones tendríamos n números. Estas son algunas de las formas de implementar en 2d un punto.

- Punto de enteros

```
struct punto { int x, y;
punto() { x = y = 0; }
punto(int _x, int _y) : x(_x), y(_y) {} };
```

- Punto de reales

```
struct punto { double x, y;
punto() { x = y = 0.0; }
point(double _x, double _y) : x(_x), y(_y) {} };
```

Operaciones con puntos

- Comparación
  Como algunos números son imposibles de representar en forma decimal por una computadora, las maquinas muchas veces aproximan el resultado y esto da lugar inprecisiones por ejemplo el numero $\frac{1}{3}$ no se puede representar en su totalidad por que tiene un número de decimales infinitos, así que cuando estamos haciendo una comparación tenemos que comparar que el valor absoluto de la resta de 2 valores es menor que ε, ε es un numero muy pequeño casi cero se define normalmente como 1e-9.

```
struct punto { double x, y;
punto() { x = y = 0.0; }
punto(double _x, double _y) : x(_x), y(_y) {}
bool operator == (punto otro) const {
return (fabs(x - otro.x) < EPS && (fabs(y - otro.y) < EPS));}};
```

- Ordenamiento
  ordenar los puntos es muy importante en el caso de que estemos buscando optimizar la busqueda de cierto punto en un arreglo, para que c++ pueda ordenar un arreglo la estructura debe tener definido el operador $<$ vamos a comprar por la coordenada x y en caso de empate compararemos la ordenada y

```
struct punto { double x, y;
punto() { x = y = 0.0; }
punto(double _x, double _y) : x(_x), y(_y) {}
bool operator < (punto otro) const {
if (fabs(x - otro.x) > EPS) return x < otro.x;
return y < otro.y; } };
sort(P.begin(), P.end()); //ordenar existiendo el vector P
```

- Distancia euclídea
  C++ tiene ya una función implementada para hallar la hipotenusa de un triangulo de rectángulo y es hypot y la usamos como muestra la imagen 3.1

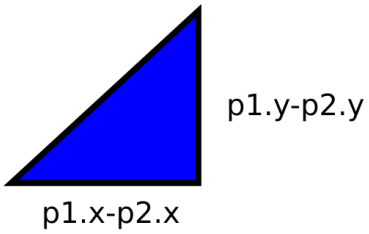


Figura 3.1: triangulo-cartesiano.png

```
double dist(punto p1, punto p2) {
return hypot(p1.x - p2.x, p1.y - p2.y);}
```

3.1.2. Lineas

Una linea es un elemento matemático que tiene infinitos puntos, una sola dimensión y va en ambos sentidos,

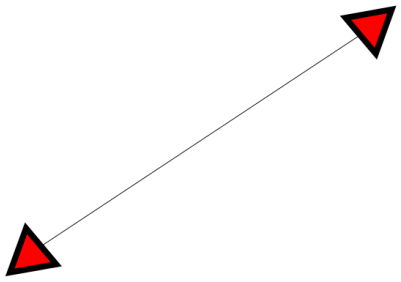


Figura 3.2: linea.png

recomendamos usar la forma $ax + by + c = 0$ y no $y = mx + b$ por que la primera tiene la capacidad de representar lineas verticales.

```
struct linea { double a, b, c; };
```

Operaciones con lineas

- hallar una linea con 2 puntos

```
    void CrearLinea(punto p1, punto p2, linea &l) {
    if (fabs(p1.x - p2.x) < EPS) {
    //Si las x son iguales es una linea vertical
      l.a = 1.0;
      l.b = 0.0;
      l.c = -p1.x;
    } else {
      l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
      l.b = 1.0;
      l.c = -(double)(l.a * p1.x) - p1.y;
    }
    }
```

- saber si dos lineas son paralelas

```
    bool sonParalelas(linea l1, linea l2) {
    return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }
```

- saber si 2 lineas son iguales

```
    bool sonIguales(line l1, line l2) {
    return sonParalelas(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }
```

- intersección entre 2 lineas

```
    bool interseccion(linea l1, linea l2, punto& p)
    {
        if (sonParalelas(l1, l2))
            //Si son paralelas la lineas no se interceptan
            return false;
        //Resolvemos el sistema de ecuaciones con dos incognitas para
           hallar la x
        p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
        /* Es posible que una de nuestras rectas sea una recta vertical
           asi que no  podremos remplazar en ella nuestra x para hallar la
            y */
        if (fabs(l1.b) > EPS)
            p.y = -(l1.a * p.x + l1.c);
        else
            p.y = -(l2.a * p.x + l2.c);
        return true;
    }
```

3.1.3. vectores

Un vector es un segmento de linea que tiene magnitud y dirección, los vectores son representados parecido a como se representa un punto con dos coordenadas x, y donde con eso ya tenemos la magnitud y dirección del vector en posición estándar.

```
struct vec { double x, y;
vec(double _x, double _y) : x(_x), y(_y) {} };
```

Si tenemos un vector que no esta en posición estándar tenemos 2 puntos *cola* y *cabeza* donde para trasformarlo a posición estándar solo tenemos que restar la cola con la cabeza.

```
vec vecAEstandar(punto cola, punto cabeza) {
return vec(cabeza.x - cola.x, cabez.y - cola.y); }
```

Operaciones con vectores

- Escalar
  Es tener un vector con una magnitud igual a la que tenia multiplicado por un numero real positivo s con la misma dirección.

```
  vec escalar(vec v, double s) {
    return vec(v.x * s, v.y * s);
  }
```

- Cuadrado de la magnitud
  Como un vector es un segmento de linea su magnitud se puede hallar con la formula de la distancia euclídea, si no sacamos la raíz tenemos la magnitud al cuadrado

```
  double cuadradoMagnitud(vec v) { return v.x * v.x + v.y * v.y; }
```

- Producto punto
  El producto punto es una operación entre vectores donde el resultado es un escalar

```
  double ProductoPunto(vec a, vec b) { return (a.x * b.x + a.y * b.y); }
```

- Producto cruz
  Normal mente el producto cruz entre 2 vectores nos da otro vector, pero a nosotros solo nos interesa la magnitud por sus aplicaciones al plano 2d como el área del paralelogramo formado por 2 vectores. la magnitud del producto cruz la podemos aya de la siguiente manera

```
  double productoCruz(vec a, vec b) { return a.x * b.y - a.y * b.x; }
```

Aplicaciones de los vectores

- Área de un paralelogramo

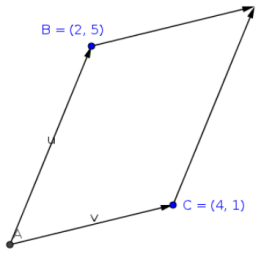


Figura 3.3: paralelogramo.png
$(2, 5)$ y $(4, 1)$

- Saber si un punto esta a la derecha o la izquierda de una recta o esta dentro de la recta
  El producto punto se pude escribir también como $\sin(\Theta)\,|a|\,|b|$

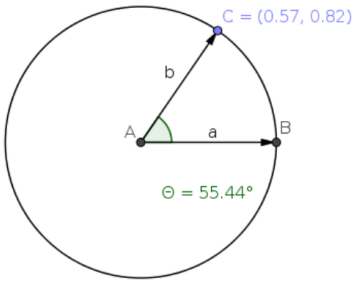


Figura 3.4: dercha-izquierda-colineal.png
Si el punto esta a la izquierda el seno del angulo sera positivo, si esta a la derecha sera negativo y si es lineal sera 0.

```
1    bool ccw(point a, point b, point c) { //confirmar si esta en sentido
        antihorario
2    return productoCruz(vecAEstandar(a, b), vecAEstandar(a, c)) > 0; }
3    bool colineal(punto a, punto b, punto c) {
4    return fabs(productoCruz(vecAEstandar(a, b), vecAEstandar(a, c))) <
        EPS; }
```

3.2. Polígonos

3.2.1. Representación

Un polígono es una región en un plano limitado por 3 o mas segmentos de linea

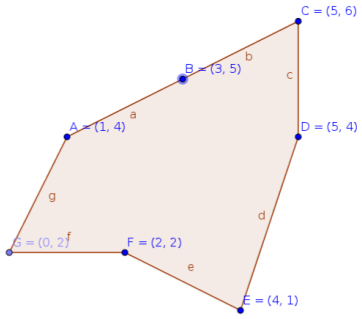


Figura 3.5: poligono.png

la forma de representar un polígono es con un arreglo de puntos de la siguiente forma:

```
vector<point> P;
P.push_back(point(1, 4)); // A
P.push_back(point(3, 5)); // B
P.push_back(point(5, 6)); // C
P.push_back(point(5, 4)); // D
P.push_back(point(4, 1)); // E
P.push_back(point(2, 2)); // F
P.push_back(point(0, 2)); // G
P.push_back(P[0]); // para cerrar el ciclo
```

3.2.2. Perímetro

El perímetro es la suma de los lados de un polígono es posible hallarlo solo recorriendo el arreglo que lo representa hallando la distancia euclídea entre el punto en el que estamos y el siguiente

```
double perimetro(const vector<punto> &P) {
  double resultado = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++) result += dist(P[i], P[i+1]);
  return resultado;
}
```

3.2.3. Área

El área se calcula como el un medio determinante de la matriz creada por los puntos que componen el polígono.

Para nuestro ejemplo tenemos: $A = \frac{1}{2} * \begin{vmatrix} 1 & 4 \\ 3 & 5 \\ 5 & 6 \\ 5 & 4 \\ 4 & 1 \\ 2 & 2 \\ 0 & 2 \end{vmatrix}$

```
double area(const vector<punto> &P) {
  double result = 0.0, x1, y1, x2, y2;
  for (int i = 0; i < (int)P.size()-1; i++)
  result += (P[i].x * P[i+1].y - P[i+1].x * P[i].y);
}
return fabs(result) / 2.0; }
```

3.2.4. Comprobar si un polígono es convexo

Para comprobar si un polígono es convexo solo tenemos que comprobar si cogiendo todos los lados de un polígono el siguiente punto que le sigue en sentido horario siempre esta a la derecha o siempre esta a la izquierda.

```
bool esConvexo(const vector<punto> &P) {
int tamano = (int)P.size();
if (tamano <= 3) return false; //los puntos o lineas no son convexos
bool primero = ccw(P[0], P[1], P[2]);
for (int i = 1; i < tamano - 1; i++)
if (ccw(P[i], P[i+1], P[(i+2) == tamano ? 1 : i+2]) != primero)
return false;
return true; }
```

este código no funciona si el polígono tiene puntos colineales.

Capítulo 4

Estructura de datos

4.1. Descripción y Motivación

Una estructura de datos es la manera en la cual se organiza la información, por esta razon es posible que este capítulo sea el que más uses en tu vida cotidiana como programador.
Comencemos imaginando dos bibliotecas, la primera es muy estricta con sus reglas y todas las personas que leen un libro, deben regresarlo a su ubicación. En cambio la segunda biblioteca no tiene un orden, los libros estan regados por todas partes y las personas que los utilizan los dejan tirados donde sea. A primera vista pareciera que la segunda biblioteca no sirve para nada, pero en realidad si tu solo deseas ir a leer cualquier cosa y luego no tener que preocuparte de donde dejar el libro la segunda biblioteca seria ideal. A lo que quiero llegar es que hay distintas formas de ordenar la información, y algunas sirven para mejorar el desempeño en algunas areas sacrificando otras, no existe una estructura perfecta que haga bien todo al mismo tiempo.
Las principales operaciones sobre las estructuras son:

- Insertar
- Buscar
- Borrar
- Actualizar

Conocer las principales estructuras de datos y entender muy bien el problema al que nos enfrentemos seran la clave para idear una solución optima.

4.2. Complejidad

No es la intención de este libro dar una explicación detallada de lo que es la complejidad de algoritmos, solo daremos una descripcion por encima de la notación big O. Esta notación nos dice cuantas ejecuciones realizaria un algoritmo en el peor de los casos, por ejemplo si tenemos que buscar un libro dentro de la biblioteca desordenada, la complejidad seria big $O(n)$ siendo n la cantidad de libros, ya que en el peor de los casos tendriamos que buscar uno por uno hasta el último libro.

4.3. Estructuras de datos lineales

Una estructura de datos es considerada lineal si todos sus elementos estan organizados en linea, por ejemplo en un arreglo de izquierda a derecha.
En la mayoria de lenguajes de programación podemos distinguir entre arreglos estaticos y arreglos dinamicos, a los arreglos estaticos les definimos un tamaño y es inalterable.

Listing 4.1: arregloEstatico.cpp

```
int main(){
    string palabras[] = {"hola","adios","tres"};
    cout<<palabras[2]<<endl;
}
```

Los arreglos comienzan con el indice 0 siendo palabras[0] = “hola”, palabras[1] = “adios” y palabras[2]=“tres”.

4.3.1. Arreglos dinámicos

Los arreglos estaticos son muy utiles cuando sabemos exactamente el tamaño de elementos que usaremos, su complejidad en las diferentes operaciones es:

- Insertar/Actualizar $O(1)$ si conocemos la casilla donde insertaremos o actualizaremos, si no $O(n)$
- Buscar $O(1)$ (cuando conocemos el indice), si no $O(n)$
- Borrar $O(1)$ o $O(n)$ esta es una operacion complicada, ya que al borrar un elemento dejamos el espacio vacio, y lo más tipico seria correr todos los elementos de la derecha a la izquierda

Para entender un poco más esto imaginemos una estanteria de libros, donde solo caben 10 libros. Esta vacia y podemos empezar a meter libros donde queramos, pero si no tenemos un orden a la hora de ponerlos cuando esta más llena nos tomara más tiempo encontrar un espacio vacio, en cambio si vamos metiendo en orden siempre sabremos donde meter el proximo. La operación de buscar seria similar a agarrar el libro de la estanteria, si sabemos exactamente donde esta solo debemos tomarlo y ya, si no empezar a mirar uno por uno hasta encontrar el que buscamos, la operación de borrar es muy simple si solo quitamos el libro, pero hay dos cosas que podrian complicarla, la primera seria saber que libro quitaremos y la segunda si queremos que no quede el espacio vacio, pues nos tocaria correr todos los libros de la derecha hacia la izquierda para llenar el agujero. La operación de actualizar sera como una mezcla entre borrar e insertar.
Pero no nos asustemos, para usos prácticos es muy simple, solo usaremos arreglos estaticos para guardar información que recorreremos completa a menudo, por ejemplo si tenemos muchos amigos y a todos les queremos dar regalos:

Listing 4.2: arregloAmigos.cpp

```
int main(){
    string amigos[5] = {"ana","brian","cesar","daniel","eliana"};
    string regalos[3] = {"abrazo","reloj","perfume"};

    for(int i=0;i<5;i++){
        for(int j=0;j<3;j++){
            cout<<"le regalo un "<<regalos[j]<<" a "<<amigos[i]<<endl;
        }
    }
}
```

4.3.2. Arreglos esáticos

Los arreglos dinamicos son iguales a los estaticos, excepto por que pueden agrandarse todo lo que quieran (siempre que lo soporte la RAM), otra gran diferencia es que ya traen por defecto la implementación de inserción y eliminación, esta estructura no permite huecos, por lo que su complejidad es la siguiente:

- Insertar $O(1)$
- Buscar $O(1)$ (cuando conocemos el indice), si no $O(n)$

- Borrar $O(n)$
- actualizar $O(1)$ (cuando conocemos el indice), si no $O(n)$

Como podemos observar, sus complejidades son muy efectivas, y por eso son muy usadas en la mayoria de las ocasiones, de hecho casi cualquier problema que requiera estructura de datos se puede solucionar aplicando esta estructura, solo que obviamente no siempre es la solución óptima. Supongamos una base de datos que solo usara arreglos, seria muy lenta y poco práctica. Un ejemplo de uso de arreglo dinámico es el siguiente:

Listing 4.3: arregloDinamicoAmigos.cpp

```
int main(){
    vector<string> amigos;
    vector<string> regalos;
    string amigo,regalo;
    cout<<"ingrese todos sus amigos, uno por uno , si ya termino ingrese 0"
       <<endl;
    while(cin>>amigo){
        if(amigo=="0")break;
        amigos.push_back(amigo);
    }
    cout<<"ingrese todos los regalos, uno por uno , si ya termino ingrese 0"
       <<endl;
    while(cin>>regalo){
        if(regalo=="0")break;
        regalos.push_back(regalo);
    }

    for(int i=0;i<amigos.size();i++){
        for(int j=0;j<regalos.size();j++){
            cout<<"le regalo un "<<regalos[j]<<" a "<<amigos[i]<<endl;
        }
    }
}
```

Generalmente la unica manera de conocer el indice del elemento que estamos buscando, es que queramos recorrer el arreglo, como lo hemos hecho en los ejemplos. Asi que en la mayoria de ocasiones cuando buscamos un único elemento la complejidad es de $O(n)$, pero podemos mejorar esto, ordenando el arreglo. Como en el ejemplo de la biblioteca tener la información ordenada nos permite encontrar las cosas más rapidamente, pero sacrificamos otras cosas a cambio. Como ya lo mencionamos en estructuras de datos no hay nada perfecto para todo, tenemos dos opciones. La primera es ordenar el arreglo antes de hacer la consulta, la otra es siempre tenerlo ordenado. Ordenar un arreglo no es una tarea fácil, por suerte la mayoria de lenguajes de programación nos provee herramientas para hacer esto, los mejores algoritmos de ordenamiento genericos tienen una complejidad de $O(n \log n)$, y buscar un elemento en un arreglo ordenado nos toma $O(\log n)$ por medio de busqueda binaria, la busqueda binaria funciona parandonos en la mitad, decidiendo si el elemento que buscamos se encuentra hacia la derecha o hacia la izquierda (lo sabemos por que estan ordenados) y repitiendo el proceso.

Figura 4.1: busquedaBinaria.png

Por ejemplo si tenemos un directorio de teléfonos y estamos buscando el número de “Sofia”, si nos paramos en la mitad del directorio encontraremos quizas las palabras que inician en “M”, sabemos

que el número que buscamos se encuentra hacia la derecha del directorio por que la “S” es mayor a la “M” asi que de una sola busqueda ya descartamos la mitad de las opciones, luego repetimos el proceso parandonos en la mitad del directorio que nos queda y esta vez nos paramos en la letra “S”, la palabra “Sofia” se encuentra en esta letra asi que nos ahorramos recorrer una por una desde la “A” hasta la “S” para encontrar la pagina que buscabamos.
Es ineficiente ordenar un arreglo para hacer una única búsqueda, pero se vuelve efectivo a partir de una cantidad, vamos a calcular en que momento se vuelve efectivo: S busquedas en un arreglo desordenado tiene una complejidad de $O(S \times n)$ y S busquedas en un arreglo ordenado tiene una complejidad de $O(n \log n + S \log n)$. Si igualamos y despejamos S, obtenemos $S = \frac{nlog(n)}{n-log(n)}$ por lo tanto si nuestra cantidad de busquedas es mayor a S, vale la pena ordenar el arreglo antes de realizarlas.
Algunas implementaciones especiales que son las pilas y las colas, estas estructuras no suelen recorrersen, en cambio se usan para ingresar elementos y retirarlos con una complejidad de $O(1)$, comencemos por las colas. Funcionan igual que una cola en un restaurante, las nuevas personas que van llegando se hacen al fondo, y deben esperarsen a que atiendan a todas las que habian llegado antes que ella. Al contrario las pilas funcionan al revés, imaginemos una pila de platos, se van lavando los que estan más arriba y el ultimo que se lava es el de más abajo, si llega un nuevo plato se pone en la cima y se lava de primero.

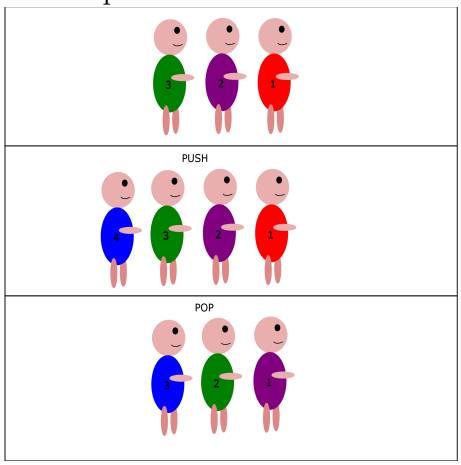


Figura 4.2: cola.png

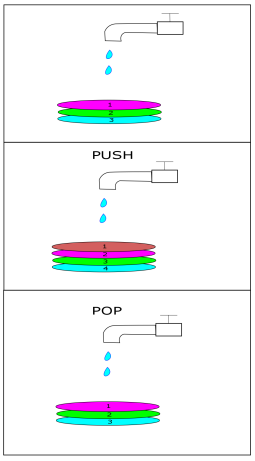


Figura 4.3: pila.png

Estos códigos de uso de colas y pilas fueron tomados de cplusplus y se pueden encontrar en esta url `http://www.cplusplus.com/reference/`.

Listing 4.4: cola.cpp

```
int main(){
    queue<int> myqueue;
    int myint;
    cout << "Please enter some integers (enter 0 to end):\n";
    do {
        cin >> myint;
        myqueue.push (myint);
    } while (myint);
    cout << "myqueue contains: ";
    while (!myqueue.empty())
    {
        cout << ' ' << myqueue.front();
        myqueue.pop();
    }
    cout << '\n';
    return 0;
}
```

Listing 4.5: pila.cpp

```
int main(){
    std::stack<int> mystack;

    for (int i=0; i<5; ++i) mystack.push(i);

    std::cout << "Popping out elements...";
    while (!mystack.empty())
    {
        std::cout << ' ' << mystack.top();
        mystack.pop();
    }
    std::cout << '\n';

    return 0;
}
```

4.4. Estructuras de datos no lineales

A veces las estructuras de datos lineales no son lo suficientemente eficientes para el enfoque de nuestro problema, para estos casos es probable que requiramos una estructura de datos no lineales.

4.4.1. Árbol binario balanceado

Un árbol binario balanceado es aquel que la altura de los hijos de cualquier nodo difieren en maximo 1. Los conjuntos y los mapas son codificados con esta estructura, entenderla y programarla es algo tedioso, pero los principales lenguajes de programación ya la traen implementada por defecto, si deseas conocer más acerca de esta estructura puedes consultarla por su nombre en español o la documentación en ingles como “Balanced Binary Search Tree”, con una complejidad en todas sus operaciones de $O(log(n))$

4.4.2. Conjuntos

Los conjuntos son muy útiles cuando se quiere preguntar si un elemento existe en el conjunto, si usaramos una estructura lineal nos tomaria $O(n)$ saber si el elemento existe. Si se intenta insertar un elemento repetido no pasa nada.

Listing 4.6: conjunto.cpp

```
int main(){
    set<string> palabrasFavoritas;
    string palabra;
    cout<<"inserte una palabra a tus favoritas, escriba 0 para terminar"<<
       endl;
    cin>>palabra;
    while(palabra!="0"){
        palabrasFavoritas.insert(palabra);
        cout<<"inserte una palabra a tus favoritas, escriba 0 para terminar"
           <<endl;
        cin>>palabra;
    }
    cout<<"pregunte por una palabra para saber si esta entre tus favoritas,
       escriba 0 para terminar"<<endl;
    string pregunta;
    cin>>pregunta;
    while(pregunta!="0"){
        if(palabrasFavoritas.count(pregunta)){
            cout<<"esta entre las favoritas"<<endl;
        }else{
            cout<<"no esta entre las favoritas"<<endl;
        }
        cout<<"pregunte por una palabra para saber si esta entre tus
           favoritas, escriba 0 para terminar"<<endl;
        cin>>pregunta;
    }
}
```

Existen otras aplicaciones para los conjuntos, ya que la información en estos esta siempre ordenada, se puede simular una estructura lineal con complejidad $O(log(n))$ en todas sus operaciones, recuerdan la comparación que hicimos antes sobre S busquedas ordenando un arreglo, al tener esta otra manera de ordenar la información se vuelve aun mas complicado decidir que estructura de datos nos conviene más, pero como norma general seria asi:

- Si suelen hacersen muchas operaciones de insersión y casi ninguna de busqueda conviene más un arreglo dinámico sin ordenar nunca.
- Si suelen hacersen muchas operaciones de insersión, seguidas de muchas busquedas conviene más un arreglo dinámico ordenandolo antes de iniciar la serie de busquedas.
- Si suelen hacerse operaciones de insersión y de busquedas uniformemente, conviene más un conjunto.

Este es un claro ejemplo de por que conocer las distintas estructuras de datos nos permite optimizar nuestra solución.

4.4.3. Mapas

Los mapas son similares a los conjuntos, la única diferencia es que permiten guardar una relación entre clave− >valor, la clave suele ser un string o un entero, pero dependiendo del lenguaje de programación puede ser de cualquier tipo de dato sobreescribiendo el operador menor que <, el valor puede ser cualquier tipo de dato sin ningún impedimento. Por ejemplo si deseamos tener una registro con todos los animales y la cantidad que hemos encontrado de estos, constantemente iremos descubriendo nuevos animales y repitiendo los que ya habiamos encontrado. Si aplicamos las estructuras de datos que conociamos tendriamos que usar un arreglo dinámico con objetos que contengan el nombre del animal y la cantidad. Pero con el mapa simplemente podemos dar como clave el nombre del animal

y como valor la cantidad, asi todas las operaciones tendrian complejidad de $log(n)$. Al igual que en el ejemplo de los sets, hay situaciones donde conviene más el uso de una lista, o una lista y ordenara antes que usar mapa, pero en la mayoria de aplicaciones las insersiones y busquedas tienen un comportamiento uniforme asi que conviene más el uso de mapas en la mayoria de casos.

Listing 4.7: conjunto.cpp

```
int main(){
    set<string> palabrasFavoritas;
    string palabra;
    cout<<"inserte una palabra a tus favoritas, escriba 0 para terminar"<<
       endl;
    cin>>palabra;
    while(palabra!="0"){
        palabrasFavoritas.insert(palabra);
        cout<<"inserte una palabra a tus favoritas, escriba 0 para terminar"
           <<endl;
        cin>>palabra;
    }
    cout<<"pregunte por una palabra para saber si esta entre tus favoritas,
       escriba 0 para terminar"<<endl;
    string pregunta;
    cin>>pregunta;
    while(pregunta!="0"){
        if(palabrasFavoritas.count(pregunta)){
            cout<<"esta entre las favoritas"<<endl;
        }else{
            cout<<"no esta entre las favoritas"<<endl;
        }
        cout<<"pregunte por una palabra para saber si esta entre tus
           favoritas, escriba 0 para terminar"<<endl;
        cin>>pregunta;
    }
}
```

4.4.4. Conjuntos disjuntos

Conocida en ingles como (Union-Find Disjoint Sets) es una estructura optimizada para tener varios conjuntos y poder ejecutar algunas operaciones casi en tiempo lineal $\approx O(1)$, en realidad la complejidad de M operaciones en esta estructura tiene una complejidad de $M * \alpha(n)$ donde n es la cantidad de elementos en todos los conjuntos, y $\alpha(n)$ es la función inversa de ackerman, la función de ackerman crece muy rapido y como efecto su función inversa crece excecivamente lento. Por esto se puede considerar $\alpha(n)$ como una constante y las M operaciones contarian con una complejidad $\approx O(M)$.
Las operaciones son:

- $Consultar(elemento)$: retorna el conjunto al que pertenece $elemento$.
- $Evaluar(elemento1, elemento2)$: evalua si $elemento1$ esta en el mismo conjunto que $elemento2$.
- $Unir(elemento1, elemento2)$: une el conjunto que contiene a $elemento1$ con el conjunto de $elemento2$.

Para lograr esta eficiencia, esta estructura reune todos los elementos en un árbol, de manera que el ancestro del arbol es el conjunto al que pertenecen, cuando un elemento e_1 se encuentra por debajo del segundo nivel del árbol (siendo el primer nivel el ancestro y el segundo nivel sus hijos directos), y se ejecuta $Consultar(e_1)$ esto tomara algunas ejecuciones hasta encontrar su ancestro, pero recursivamente iremos estableciendo al ancestro como padre directo del elemento e_1 y de todos sus padres.

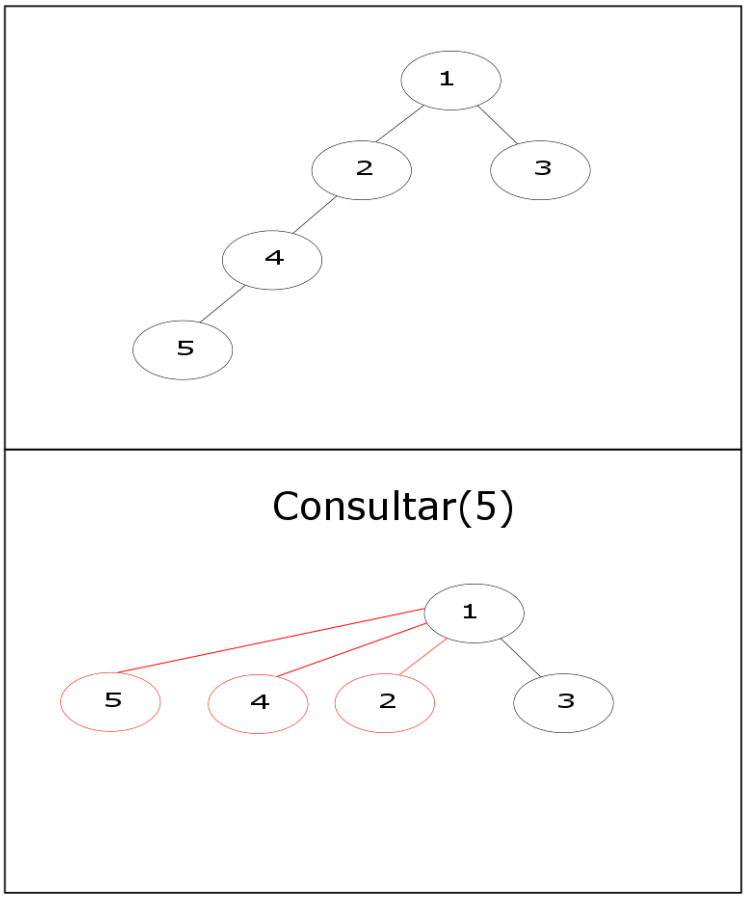


Figura 4.4: conjuntosDisjuntosConsultar.png

Si queremos unir los conjuntos de dos elementos *elemento*1 y *elemento*2, lo único que debemos hacer es consultar el ancestro de ambos elementos: $Consultar(elemento1)$ y $Consultar(elemento2)$, si son distintos (pertenecen a diferente conjunto) asignamos a uno de los dos ancestros como padre del otro.

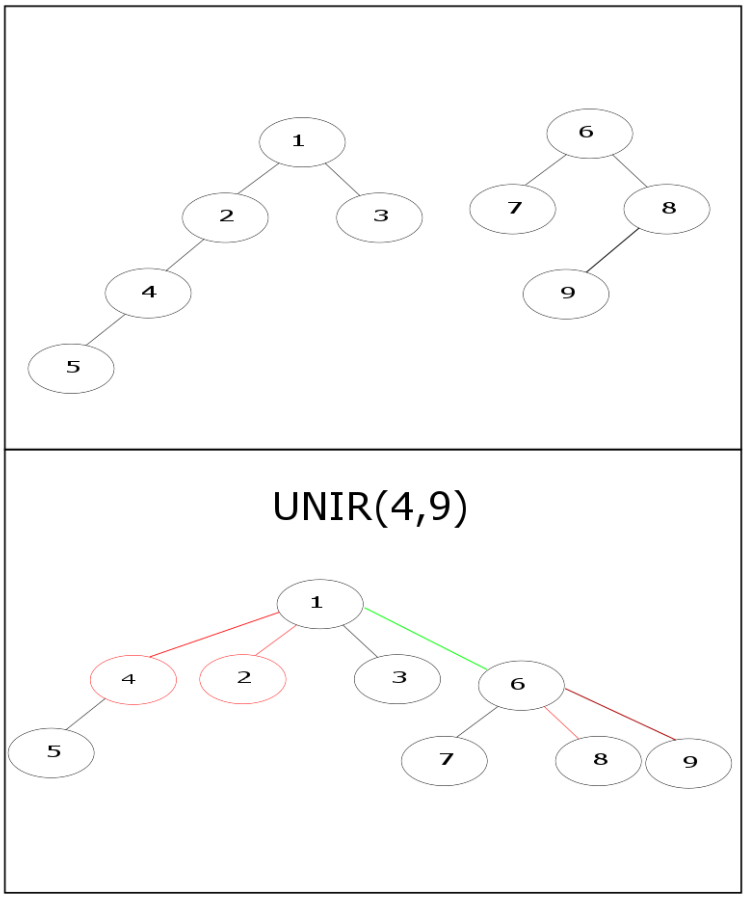


Figura 4.5: conjuntosDisjuntosUnir.png

Para evaluar si dos elementos pertenecen al mismo conjunto unicamente debemos comparar los resultados entre $Consultar(elemento1)$ y $consultar(elemento2)$.

Podemos observar que el metodo *consultar* actualiza el árbol cada que se ejecuta haciendo que el elemento y sus padres esten a solo un nodo de distancia del ancestro, y tanto $Evaluar$, como $Unir$ hacen uso de $Consultar$, es por eso que conforme se van haciendo ejecuciones, el arbol va manteniendo su tamaño reducido y sus operaciones tienden a ser $O(1)$.

Listing 4.8: unionFind.cpp

```
#include <bits/stdc++.h>
using namespace std;

//1000 es el limite de elementos, puede modificarse
vector<int> pset(1000); //padre del elemento i
void inicializarConjuntos(int N) {
    pset.assign(N, 0);
    for (int i = 0; i < N; i++){
        pset[i] = i;
    }
}
int consultar(int i) {
    if(pset[i]==i){
        return i; //si ya es el ancestro lo retorna.
    }else{
        pset[i] = consultar(pset[i]); //si no es el ancestro, hace que su
           papa sea su ancestro y lo retorna
        return pset[i];
    }
}
bool evaluar(int i, int j) {
    return consultar(i) == consultar(j); //si tienen el mismo ancestro
       retorna true, si no false.
}

void unionSet(int i, int j) {
    if (!evaluar(i, j)) { //si no son el mismo conjunto, consulta el
       ancestro de i y de j, luego hace que el padre del ancestro de i sea
       el ancestro de j
        pset[consultar(i)] = consultar(j);
    }
}

int main() {
  printf("asumimos 5 elementos en 5 conjuntos diferentes al empezar\n");
  inicializarConjuntos(5); // create 5 disjoint sets
  unionSet(0, 1);
  unionSet(0, 2);
  unionSet(3, 1);
  printf("consultar(A) = %d\n", consultar(0));
  printf("consultar(B) = %d\n", consultar(1));
  printf("consultar(C) = %d\n", consultar(2));
  printf("consultar(D) = %d\n", consultar(3));
  printf("consultar(E) = %d\n", consultar(4));
  printf("evaluar(A, E) = %d\n", evaluar(0, 4));
  printf("evaluar(A, B) = %d\n", evaluar(0, 1));

  return 0;
}
```

Este código fue tomado y modificado de [5]. Todo el capítulo fue inspirado en [6].

Capítulo 5

Programación Dinámica

5.1. Descripción y Motivación

La programación dinámica (resumido como dp) es quizás uno de los temas más complejos de tratar, por que más que teoría es casi un paradigma de programación.
Pero ¿qué es la programación dinámica?, la principal caracteristica de la programación dinámica es que se puede solucionar hallando la solucion de subproblemas, en general soluciona problemas de tipo optimización, maximización, minimización o conteo.

5.2. Memorización

La primera técnica que estudiaremos es la memorización, esta es muy util en algoritmos recursivos ya que evita que recalculemos desde una simple operacion hasta una rama completa de iteraciones. Esto se entiende mejor con un ejemplo, recordemos el algoritmo recursivo de fibbonacci.

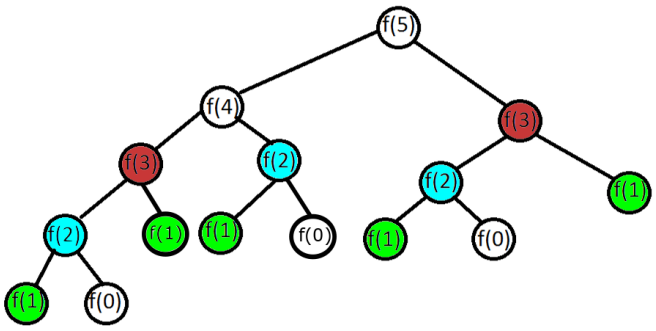


Figura 5.1: fibonacci.png

Si observamos los nodos coloreados, vemos que recalculamos mucho en especial toda la recursion de f(3) pintada en rojo, mientras más crecemos en el f(n), más grandes son los subarboles recursivos que recalculamos, es por eso que si memorizamos las soluciones solo hariamos los siguientes calculos:

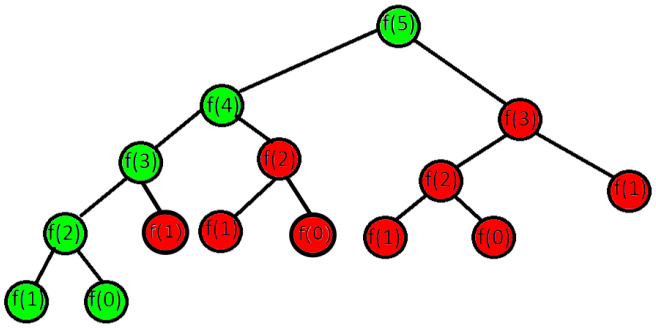


Figura 5.2: fibonacciMemorizacion.png

Solo calculamos los nodos pintados en verdes, los resultados de los nodos rojos los obtenemos por medio de la memorización, en este ejemplo nos ahorramos más de la mitad de iteraciones. Esto se lograria con un código como el siguiente:

Listing 5.1: fibonacciMemorizacion.cpp

```
int memorizacion[100];
int fibonacci(int n){
    if(n==0)return 0;
    if(n==1)return 1;
    if(memorizacion[n]==0){
        memorizacion[n] = fibonacci(n-1) + fibonacci(n-2);
    }
    return memorizacion[n];
}
```

Este código es una pequeña modificacion al fibbonacciRecursivo.cpp del capitulo de recursividad, lo único que hacemos es agregarle un arreglo en el cual memorizamos las soluciones que vamos resolviendo, es importante que la recursión tenga acceso a este arreglo, puede hacerse creando el arreglo como variable global o mandandolo como parametro, en este caso y con fines de maratones de programación se usa como variable global ya que es más facil de codificar para estas competencias, pero en proyectos siempre recomendamos seguir las buenas practicas de programación.
Otro ejemplo en el que el uso de la memorización nos puede ayudar es en hallar los coeficientes binomiales por medio de su formula recursiva (ver capitulo matematico)
$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ para todos los números enteros $n, k > 0$,
con valores iniciales
$\binom{n}{0} = 1$ para todos los números enteros $n >= 0$,
$\binom{0}{k} = 0$ para todos los números enteros $k > 0$.
Por ejemplo $\binom{3}{2}$ funciona asi:

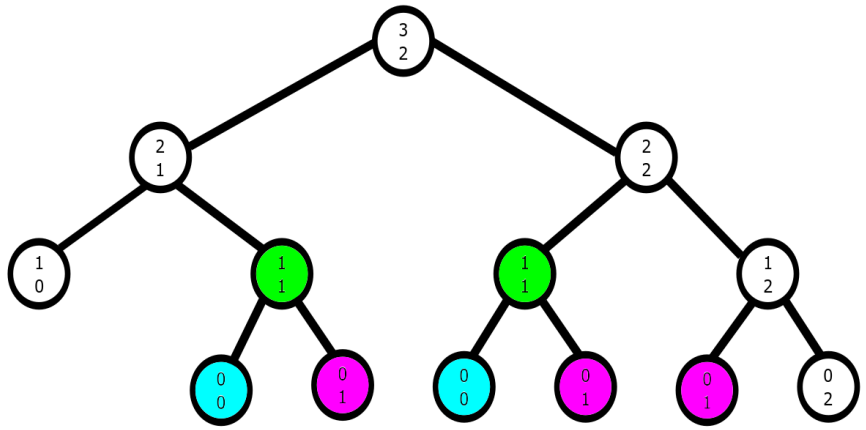


Figura 5.3: combinatoria.png

Podemos observar que con memorización nos ahorramos algunas repeticiones, pero si examinamos con detenimiento el crecimiento de la recursión y de la memorización, podemos observar lo siguiente: El crecimiento de la recursión es casi exponencial, ya que casi cada nodo se divide en 2, generando una complejidad un poco menor que $O(2^n)$, sin embargo el crecimiento de la memorización es de n^2 ya que si tenemos $\binom{n}{m}$ necesitariamos poder almacenar unicamente las combinaciónes desde 0 hasta n contra las combinaciones desde 0 hasta m.

Listing 5.2: coeficientesBinomiales.cpp

```
int memorizacion[100][100];
int coeficienteBinomial(int n,int k){
    //casos base
    if(n>=0 && k==0)return 1;
    if(k>0 && n==0)return 0;
    if(memorizacion[n][k]==0){
        memorizacion[n][k] = coeficienteBinomial(n-1,k-1)+
           coeficienteBinomial(n-1,k);
    }
    return memorizacion[n][k];
}
```

Hay varias cosas ha tener en cuenta cuando se use esta técnica, la primera son los limites del arreglo de memorización, en este caso creamos un arreglo de $100 * 100$, lo cual nos condiciona a no poder calcular un n o un k superior a 100, la decisión entre usar memorización y no usarla, dependera del poder de computo y almacenamiento que se desea ocupar en el algoritmo. También se debe tener en cuenta los estados posibles de la solución, en los dos ejemplos preguntamos si una casilla en el arreglo es igual a 0 para guardar una nueva solución,$memorizacion[n] == 0$ y $memorizacion[n][k] == 0$, pero hay recursiones en las que valores positivos, negativos, cero y hasta objetos sean resultados válidos, en estos casos se puede utilizar un arreglo booleano extra que indique si esta solución ya ha sido calculada antes.

5.3. Problemas clásicos

5.3.1. Problema de la mochila

Dado un conjunto de objetos,con su valor y peso. Determine el valor máximo que puedes cargar en una mochila que soporta w peso máximo. Este problema ha sido muy importante en las ciencias de la computación, por ser un problema NP-Completo. Si deseas ahondar más en el tema, puedes encontrar mucha más información, al momento de escribir el capitulo recomendaria mucho más la información en ingles, buscando como “knapsack problem”. Existen variantes al problema, utilizaremos la más común que es “problema de la mochila 0-1”, consiste en que solo se puede llevar una copia de cada objeto. Para solucionar el problema, lo primero que se nos podria ocurrir es evaluar todas las posibilidades, esto nos

dara la respuesta optima, pero solo podremos usarla en casos muy pequeños ya que su complejidad es de $O(2^n)$. Otra solución rápida seria ordenar los objetos de menor a mayor tamaño o de mayor a menor valor, y empezar a introducirlos hasta que no quepan más, pero esta solución no nos dara el valor máximo posible.
Existe una solución "lineal" para el problema de la mochila, más adelante explicare por que pongo entre comillas lineal. Como la solución de problemas de programación dinámica requiere solucionar subproblemas, muchas veces podemos plantarnos la idea de que pasaria si tuvieramos la respuesta a un subproblema para hallar la solución de este y asi mismo la solución del subproblema es la solución del subproblema del subproblema, frenemos antes de que explote nuestro cerebro y vamos a solucionar el problema de la mochila, podemos suponer que nos falta únicamente decidir si introducir o no un objeto, y que conocemos el valor máximo a cualquier capacidad $<= w$ de mochila. Por ejemplo tenemos una mochila que le caben 10 kilogramos, ya hemos calculado su valor máximo hasta el momento. Pero nos dimos cuenta que nos falto analizar el objeto o_i un oso de 3 kilogramos, ahora tenemos dos opciones para maximizar nuestro valor: dejarla tal cual como esta, o liberar 3 kilogramos y introducir el oso (liberar 3 kilogramos puede consistir en vaciar toda la mochila y llenar 7 kilogramos con otras cosas más optimas). Hacer este proceso nos garantiza la solución optima entre introducir o no el oso de 3 kilogramos, si profundizamos el subproblema seria maximizar el valor que le cabria a una mochila de 7 kilogramos con el objeto o_{i-1}.
Dado $dp[i][j]$ el valor máximo con los elementos $[1, 2, 3..., i]$ en una mochila de capacidad j,$w[i]$ el peso del o_i objeto y v[i] el valor del o_i objeto la formula que resuelve este problema es:

$$dp[0][j] = 0$$

$$dp[i][j] = max \left\{ \begin{array}{l} dp[i-1][j] \\ dp[i-1][j-w[i]] + v[i] \quad \text{si } j >= w[i] \end{array} \right\}$$

Listing 5.3: mochila.cpp

```
int dp[100][100];
int w[100];
int v[100];
int mochila(int i,int j){
    //casos base
    if(i==0)return 0;
    if(dp[i][j]==0){
        dp[i][j] = mochila(i-1,j);
        if(j>=w[i]){
            dp[i][j] = max(mochila(i-1,j),mochila(i-1,j-w[i])+v[i]);
        }
    }
    return dp[i][j];
}
```

En este ejemplo usamos una entrada maxima de 100 objetos, y un tamaño maximo de la mochila de 100. Ahora la complejidad de este algoritmo es de $O(n * W)$ siendo n la cantidad de objetos y W el peso máximo de la mochila, como nota curiosa puse "lineal" entre comillas por que W no esta condicionado por la entrada que son los n objetos,por lo cual el problema sigue considerandose NP-Completo,haciendo de esta solucion inaceptable por ejemplo para ejercicios de pocos objetos con un tamaño descomunal.
Al momento de escribir este cápitulo, se podia encontrar esta estupenda calculadora del problema de mochila en este link `http://karaffeltut.com/NEWKaraffeltutCom/Knapsack/knapsack.html`. En caso de que ya no exista, pueden buscar "knapsack problem calculator" y de seguro encontraran una similar.

5.3.2. Problema de subsecuencia común más larga

El problema de subsecuencia común más larga (llamado Longest Common Subsequence o LCS en ingles) consiste en encontrar la subsecuencia más larga en común entre dos secuencias.En este

subcapitulo analizaremos todas las secuencias como Strings, pero una secuencia puede consistir en un arreglo de números, incluso hasta de objetos.E studiaremos una pequeña variante más sencilla que es hallar cuantos elementos tiene la subsecuencia común más larga, pero la solucion a hallar cual es la subsecuencia común más larga es muy similar. Una subsecuencia consiste en tomar n elementos de la secuencia en el mismo orden, estos n elementos pueden ser desde ninguno hasta todos, dos subsecuencias son comunes cuando contienen exactamente los mismos elementos en el mismo orden, y la subsecuencia común más larga es aquella que ninguna otra combinacion de las posibles subsecuencias entre las dos secuencias contenga mayor cantidad de elementos. Por ejemplo si tomamos las secuencias $a = [A, B, C, D, G, H]$ y $b = [A, E, D, F, H, R]$ la subsecuencia común mas larga es $[A, D, H]$

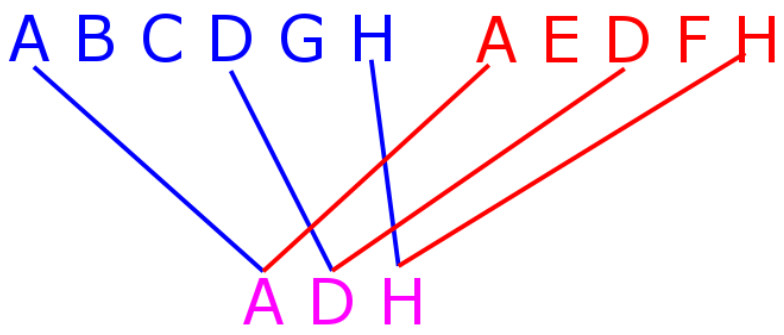


Figura 5.4: ejemploLCS.png

Podriamos considerar que una solución seria ir recorriendo la primera secuencia vs la segunda, y cuando coincidan los elementos agregarlo al LCS, pero esto no siempre dara la respuesta correcta como en este caso $a2 = [A, B, C, D]$ $b2 = [A, C, D, B]$ ya que si lo hacemos recorriendolos unicamente obtendriamos $[A, B]$. Otra solución que si daria el resultado correcto seria probar todas las combinaciones entre subsecuencias, pero esto tendria una complejidad de $O(2^{n+m})$ siendo n la cantidad de elementos de la primera secuencia y m los elementos de la segunda.
Una mejor solución aplicando dp consiste en lo siguiente: supongamos que conocemos la $LCS(i, j)$ de cualquier par de secuencias excepto la de $LCS(0, 0)$, Siendo $LCS(i, j)$ la subsecuencia común más larga entre una secuencia a y una secuencia b, haciendo un corte a las secuencias a,b en los indices i,j respectivamente quedando $LCS(i, j) = LCS(a[i : n], b[j : m])$ siendo n el último elemento de la secuencia a y m el último elemento de la secuencia b. Por ejemplo $LCS(1, 1) = [D, H]$ o $LCS(4, 1) = [H]$ o $LCS(1, 0) = [C, D]$ o $LCS(0, 1) = [C, D]$.

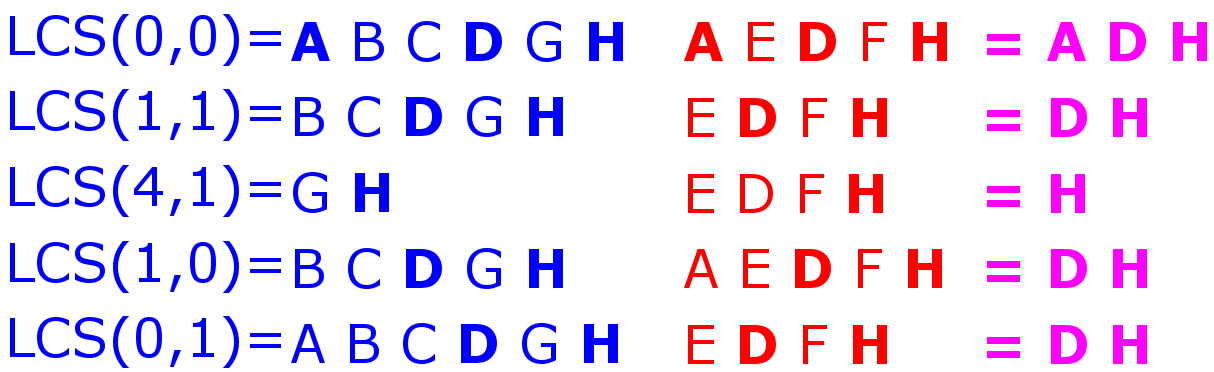


Figura 5.5: ejemplosMultiplesLCS.png

Conociendo esto intentaremos solucionear $LCS(0, 0)$ tendremos tres opciones:

- este caso solo es posible si $a_0 = b_0$ para este caso lo unico que debemos hacer es agregar el elemento al LCS y luego evaluar $LCS(1, 1)$

- descartamos completamente agregar b_0 para esto lo unico que debemos hacer es evaluar $LCS(0,1)$
- descartamos completamente agregar a_0 para esto lo unico que debemos hacer es evaluar $LCS(1,0$

Un ejemplo más enfocado para la segunda y tercera opción es este: $a3 = [A, B, C]$ $b3 = [B, C]$ como $a3_0 \neq b3_0$ debemos evaluar $LCS(0,1)$ vs $LCS(1,0)$

Figura 5.6: ejemplosMultiples2LCS.png

Si conocemos $LCS(0,1)$, $LCS(1,0)$ y $LCS(1,1)+1$ alguno de los tres debe ser la respuesta a $LCS(0,0)$ ya que las subsecuencias deben ser consecutivas, por eso al probar las 3 opciones realmente estamos probando todas las combinaciones posibles. Ahora que sabemos esto podemos imaginar que se cumple para cualquier $LCS(i,j)$ con algunas pequeñas modificaciones quedando asi:
$LCS(i, m+1) = 0$ siendo m el último elemento de b
$LCS(n+1, j) = 0$ siendo n el último elemento de a

$$LCS(i,j) = max \left\{ \begin{array}{ll} LCS(i+1,j) & \\ LCS(i,j+1) & \\ 1 + LCS(i+1,j+1) & \text{si } a_i = b_j \end{array} \right\}$$

Listing 5.4: LCS.cpp

```
int memorizacion[100][100];
int lcs(int i,int j){
    int m = b.size()-1;
    int n = a.size()-1;
    if(j==m+1)return 0;
    if(i==n+1)return 0;
    if(memorizacion[i][j]!=0)return memorizacion[i][j];
    int maximo = max(lcs(i+1,j),lcs(i,j+1));
    if(a[i]==b[j]){
        maximo = max(maximo,1+lcs(i+1,j+1));
    }
    memorizacion[i][j] = maximo;
    return memorizacion[i][j];
}
```

Aqui tambien usamos la técnica de memorización, y de hecho en casi todos los dps para no recalcular al solucionar subproblemas, ya que si no la usamos nuestro crecimiento puede llegar a ser exponencial perdiendo todo el sentido al esfuerzo realizado en la solución. En esta implementación consideramos el indice 0 como el primer elemento de la secuencia, algunas implementaciones más sencillas lo consideran el último elemento, ya depende de las preferencias del codificador.
Al momento de escribir este cápitulo, se podia encontrar esta calculadora del problema de subsecuencia común más larga en este link `http://lcs-demo.sourceforge.net/`, aun que en ella muestran el algoritmo iterativo y no recursivo, pero eso ya lo discutimos en el capitulo de recursividad, todos los algoritmos recursivos pueden ser escritos iterativamente. En caso de que ya no exista, pueden buscar "Longest common subsecuence calculator" y de seguro encontraran una similar.

5.3.3. Problema de la subsecuencia creciente más larga

Este problema también es llamado llamado (Longest Increasing Subsequence o LIS en ingles) Ya explicamos que es una subsecuencia en el problema de subsecuencia común más larga, este problema es similar, pero en vez de dos secuencias solo tenemos una y lo que debemos hallar es una subsecuencia

en la cual todos sus elementos van de menor a mayor, usaremos la variante de calcular únicamente cuantos elementos posee la subsecuencia creciente más larga, ya que esta variante es más facil, pero para saber cuales son estos elementos se hace de la misma manera con unos pasos adicionales. Formalmente dado una secuencia a hallar una subsecuencia b tal que $b_i < b_j$ y $i < j$ $\forall i, j \in a$. Por ejemplo si tenemos la secuencia $[-7, 10, 9, 2, 3, 8, 8, 1, 2, 3, 4]$ la subsecuencia creciente más larga seria $[-7, 1, 2, 3, 4]$. Al igual que el problema de subsecuencia común más larga, si intentamos resolver este problema por medio de todas las subsecuencias posibles tendriamos una complejidad de $O(2^n)$, una solución más óptima consiste en crear una copia de la secuencia a, ordenarla y aplicar subsecuencia común más larga sobre a y $a_{ordenada}$, esto nos da una complejidad cercana a $2 * n^2$, otra solución un poco más optima se consigue aplicando dp directamente sobre a sin necesidad de crear una copia.

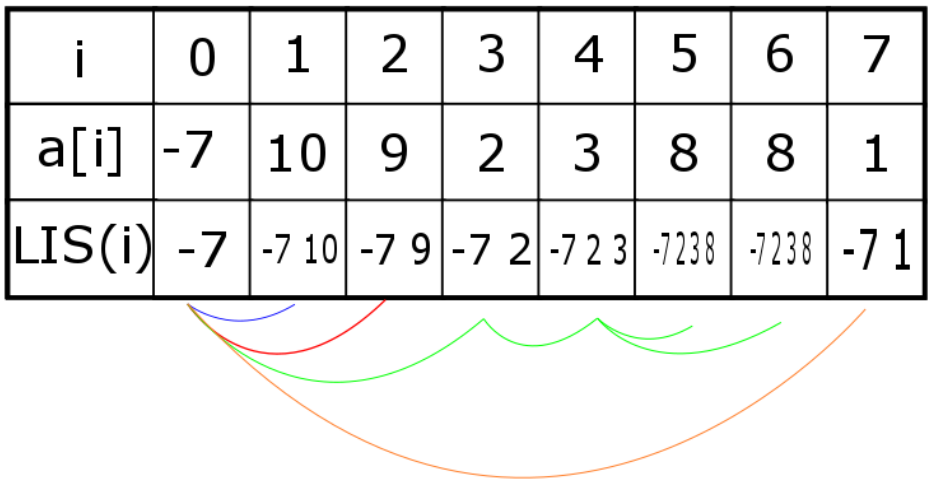


Figura 5.7: ejemploLIS.png

Si consideramos $LIS(i)$ como la solución de la secuencia desde el indice 0 hasta i, usando el elemento i aun que esta no sea la secuencia más larga como vemos en el ejemplo con $i = 7$. $LIS(i) = LIS(a[0 : i])$, teniendo esto podemos calcular $LIS(i + 1)$ de la siguiente forma:
$LIS(0) = 1$, si únicamente tenemos un elemento ese sera la subsecuencia creciente más larga.
$LIS(i + 1) = max(LIS(j) + 1), \forall j \in [0..i]$ si $a[j] < a[i + 1]$
$LIS(i+1) = 1, \forall j \in [0..i]$ si $a[j] > a[i+1]$ Esto quiere decir que si queremos incluir el elemento $a[i]$ en la solución, este debe ser el mayor de la subsecuencia ya que va al final. Hagamos paso a paso $LIS(4)$:
como $a[0] < a[4] -> LIS(4)_0 = LIS(1) + a[4] = [-7, 3]$
como $a[1] > a[4] -> LIS(4)_1 = a[4] = [3]$
como $a[2] > a[4] -> LIS(4)_2 = a[4] = [3]$
como $a[3] < a[4] -> LIS(4)_3 = LIS(3) + a[4] = [-7, 2, 3]$
Ahora solo elegimos de todas las opciones la más larga, en este caso $LIS(4)_3$, del mismo modo podemos calcular $LIS(i + 2)$,$LIS(i + 3)$...

Listing 5.5: LIS.cpp

```
int memorizacion[100];
int LIS(int i){
    if(i==0)return 1;
    if(memorizacion[i]!=0)return memorizacion[i];
    int maximo = 1;
    for(int j=0;j<i;j++){
        if(a[j]<a[i]){
            maximo = max(maximo,LIS(j)+1);
        }
    }
    memorizacion[i] = maximo;
    return memorizacion[i];
}
```

Para efectos de la demostración describi la formula recursiva con $i+1$, pero en el codigo en realidad la use de la siguiente manera:
$LIS(0) = 1$, si únicamente tenemos un elemento ese sera la subsecuencia creciente más larga.
$LIS(i) = max(LIS(j)+1), \forall j \in [0..i-1]$ si $a[j] < a[i+1]$
$LIS(i) = 1, \forall j \in [0..i-1]$ si $a[j] > a[i+1]$ esta implementación tiene una complejidad de $O(n^2)$, existe una solución a este problema de complejidad $O(n * log(k))$ siendo k la cantidad de elementos de a, que es una mejora de esta, la diferencia consiste en que no se recorre de $0...j$ si no que se mantiene un arreglo ordenado con las soluciones de $LIS(0), LIS(1)...LIS(j)$ mientras se calculan, y a la hora de revizarlos, se hace una busqueda binaria buscando el último elemento e en el arreglo l tal que $l[e] < a[j]$. En el repositorio de luisfcofv se pueden encontrar los códigos del libro Competitive Programming, dentro se encuentra la solución optimizada y que reconstruye todo el LIS, en esta url `https://github.com/luisfcofv/competitive-programming-book/blob/master/ch3/ch3_06_LIS.cpp/`.

Capítulo 6

Grafos

6.1. Descripción y Motivación

Muchos problemas de la vida real pueden ser resueltos con grafos, los grafos son conjuntos de nodos unidos por aristas, a estas aristas se les puede dar pesos u otras abstracciones necesarias, por ejemplo el mapa de una ciudad, los sitios importantes serian los nodos y las carreteras que los unen las aristas. Estas aristas podrian incluir la información de la distancia entre los nodos.

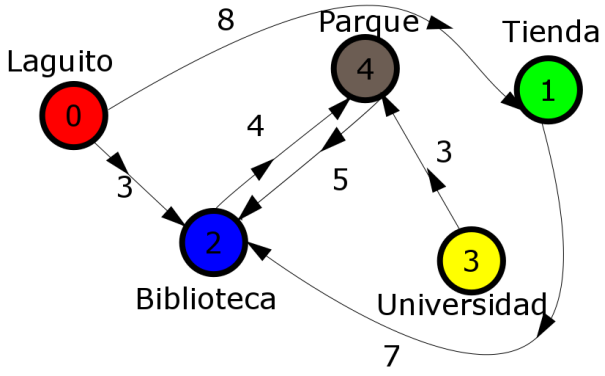


Figura 6.1: grafoEjemplo.png

Podemos observar como el nodo 0 que representa al "Laguito" esta conectado con el nodo 1 que representa la "Tienda" y se encuentra a 8 unidades de distancia, las conexiones pueden ser unidireccionales o bidireccionales, en este caso podemos ir desde el "Laguito" a la "Tienda" pero no regresar. Si bien podemos ir desde la "Biblioteca" hasta el "Parque" y regresar, son dos conexiones unidireccionales separadas. Una conexión bidireccional es una sola que puede recorrerse en ambas direcciones, como una carretera de doble via. Los grafos pueden ser representados de varias formas, las más comunes en programación son las siguientes:

- Matriz de adyacencia: La matriz de adyacencia consiste en un arreglo a de dos dimenciones en el que $a[i][j]$ representa la conexión entre el nodo i con el nodo j.

	0	1	2	3	4
0		8		3	
1			7		
2					4
3					3
4			5		

Figura 6.2: matrizDeAdyacencia.png

- Lista de adyacencia: Consiste en tener un arreglo unidimensional por cada nodo, cada elemento de este arreglo debe contener el nodo al cual establece la conexión, también puede contener la información de la conexión.

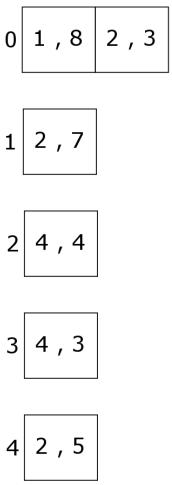


Figura 6.3: listaEnlazada.png

- Lista de aristas: Consiste en tener una unica lista unidimensional por cada conexión, cada elemento de este arreglo debe contener el nodo de origen y el nodo destino de la conexión también puede contener la información de la conexión

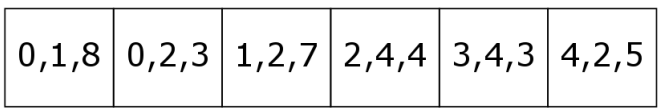


Figura 6.4: listaDeAristas.png

Cada imagen representa el mismo grafo en sus distintas formas. Que forma se elige para cual problema dependera de de que operaciones se realizaran sobre los mismos y la memoria dispuesta a ocupar, por ejemplo la matriz de adyacencia ocupa más memoria pero acceder a una conexión es instantanea pero encontrar una conexión sin conocerla para recorrer el grafo puede tomar hasta n ejecuciones, en

cambio la lista enlazada puede tomar hasta e ejecuciones encontrar una conexión en particular, pero encontrar la primera conexión para recorrer el grafo es instantaneo. Un buen simulador de grafos con sus distintas representaciones se puede encontrar aqui [4]

6.2. Recorrer un grafo

Es muy común tener que recorrer un grafo, por ejemplo si queremos saber si es posible ir desde un nodo a otro por alguna ruta.

6.2.1. busqueda en profundidad

Conocida como dfs en ingles (deep first search). A partir del nodo origen se visita su primer hijo y toda su profundidad antes de visitar sus demas hijos. Si llega a un nodo que ya ha sido visitado no lo revisita.

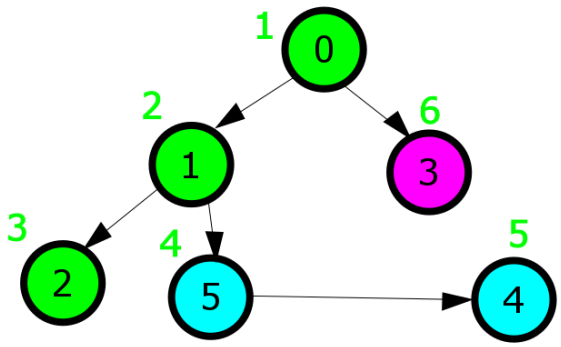


Figura 6.5: dfs.png
En este caso iniciamos la busqueda desde el nodo 0, el orden de las visitas esta en letra verde.

6.2.2. busqueda en anchura

Conocida como bfs en ingles (Breadth first search). A partir del nodo origen se visitan todos sus hijos siendo estos la primera capa, luego se visitan todos sus hijos de la primera capa siendo estos la segunda capa y asi sucesivamente.

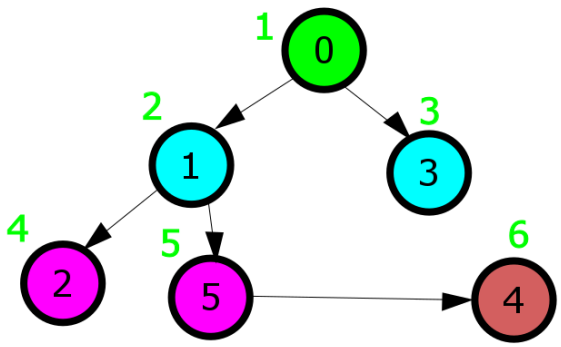


Figura 6.6: bfs.png
Un buen simulador de recorrido de grafos en sus diferentes formas se puede encontrar aqui [3]

6.3. Camino más corto desde una fuente

Para hallar el camino más corto desde una fuente usaremos el algoritmo de Dijkstra, Consiste en mantener una lista l con las distancias desde el nodo origen s, al inicio se llena con las distancias desde el nodo s hasta sus hijos, después se toma el elemento que contenga la distancia más pequeña dentro de la lista l y se elimina de la lista,este elemento contiene la distancia d_1 que es la más pequeña hasta el nodo n_1, hasta el momento es obvio que ir desde s hasta n_1 directamente es el camino más corto con una distancia de d_1. Ahora se añade a la lista las distancias de todos los destinos alcanzables desde n_1 sumada a d_1, despues se vuelve a buscar en la lista l el elemento con la distancia d_2 mas pequeña, esto nos garantiza que siempre que saquemos el elemento e_n de la lista l contendra la distancia más corta alcanzable desde s hasta n_n a menos que este nodo ya lo hallamos retirado antes de la lista l. En resumen mantenemos una lista actualizada con todas las distancias desde s a todos los nodos alcanzables por todos los caminos posibles, y la más pequeña distancia es con certeza el camino más corto desde s hasta ese nodo.

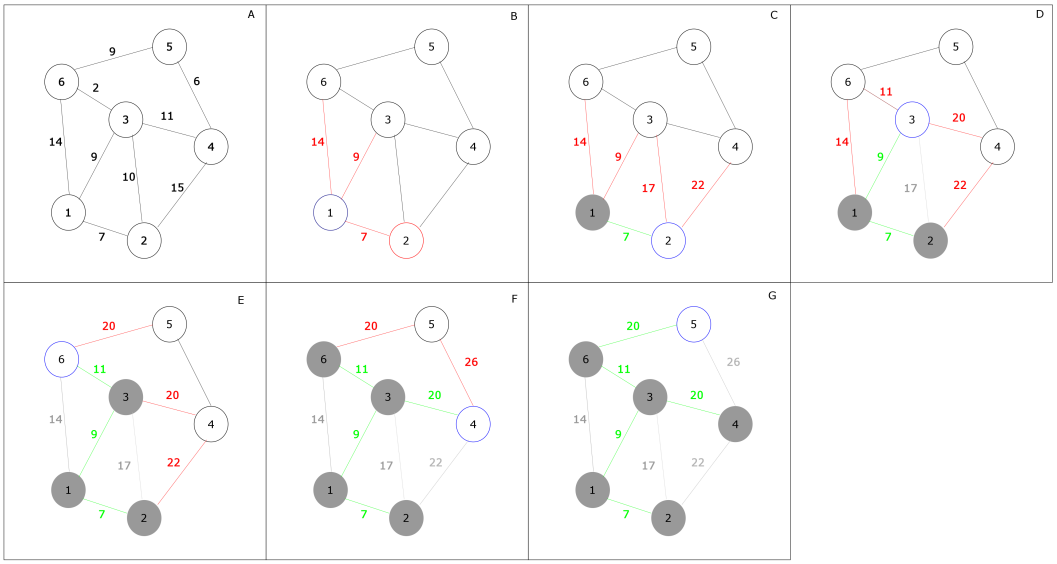


Figura 6.7: Dijkstra.png

Bibliografía

[1] Ana Echavarría, J. F. C. semillero de programación de la universidad eafit. https://github.com/anaechavarria/SemilleroProgramacion, 2013.

[2] Foundation, F. S. licencia gfdl. https://www.gnu.org/licenses/fdl-1.3.en.html, 2017.

[3] Steven, H., and Felix, H. graphdfsbfs. https://visualgo.net/en/dfsbfs.

[4] Steven, H., and Felix, H. graphds. https://visualgo.net/en/graphds.

[5] Steven, H., and Felix, H. ch208unionfindds.cpp. https://github.com/BrockCSC/acm-icpc/blob/master/resources/c%2B%2B_and_stl/ch2-Data_Structures/ch2_08_unionfind_ds.cpp, 2012.

[6] Steven, H., and Felix, H. *Competitive Programming 3*, 3 ed. lulu, 2013.

[7] uva. uva teletransport. https://uva.onlinejudge.org/external/127/12796.pdf, 2014.

[8] Visualgo. Floyd's cycle-finding. https://visualgo.net/es/cyclefinding, 2017.