

# La magia de la programación competitiva

Comunidad new liberty

2 de agosto de 2017



# Índice general

<b>Lista de figuras</b>	<b>5</b>
<b>Lista de tablas</b>	<b>7</b>
<b>1. recursividad</b>	<b>9</b>
1.1. Descripción y Motivación . . . . .	9
1.2. Ejemplos . . . . .	11
<b>2. matemáticas</b>	<b>17</b>
2.1. sucesiones y series . . . . .	17
2.1.1. sucesión aritmética . . . . .	17
2.1.2. sucesión geométrica . . . . .	17
2.1.3. serie aritmética . . . . .	17
2.1.4. serie geométrica . . . . .	17
2.2. sumatorios . . . . .	17
2.3. teoria de numeros . . . . .	18
2.3.1. aritmetica modular . . . . .	18
2.3.2. máximo común divisor y minino común múltiplo . . . . .	18
2.3.3. algoritmo de Euclides extendido . . . . .	18
2.3.4. ecuaciones diofanticas lineales de 2 variables . . . . .	18
2.3.5. conjunto $z_n$ . . . . .	18
2.3.6. propiedades de la aritmética modular . . . . .	18
2.3.7. inverso multiplicativo en $z_n$ . . . . .	19
2.4. combinatoria . . . . .	19
2.4.1. principio multiplicativo . . . . .	19
2.4.2. numero de permutaciones de n elementos . . . . .	19
2.4.3. numero de permutaciones de n elementos tomados de a m . . . . .	19
2.4.4. número de permutaciones de n elementos tomados de a m con repetición . . . . .	19
2.4.5. número de permutaciones con al menos un elemento fijo . . . . .	19
2.4.6. numero de permutaciones donde el primer elemento se repite a veces el segundo b veces ... . . . .	20
2.4.7. número de desarreglos . . . . .	20
2.4.8. número de permutaciones de n elementos que dejan exactamente k elementos fijos	20
2.4.9. número de combinaciones de n elementos tomados de a m . . . . .	20
2.5. probabilidad . . . . .	20
2.5.1. regla de Laplace . . . . .	20
2.5.2. probabilidad de intersección de sucesos . . . . .	20
2.5.3. probabilidad de unión de sucesos . . . . .	20
2.5.4. probabilidad condicionada . . . . .	20
2.5.5. teorema de Bayes . . . . .	21
2.6. transformaciones lineales . . . . .	21

<b>3. geometricos</b>	<b>23</b>
3.1. formulas de geometría . . . . .	23
3.2. estructuras geométricas . . . . .	23
3.2.1. puntos . . . . .	23
3.2.2. líneas . . . . .	24
<b>4. Estructura de datos</b>	<b>27</b>
4.1. Descripción y Motivación . . . . .	27
4.2. Complejidad . . . . .	27
4.3. Estructuras de datos lineales . . . . .	27

# Índice de figuras

1.1. fibonacci.png . . . . .	11
1.2. torre1.png . . . . .	13
1.3. torre2.png . . . . .	13
1.4. torre3.png . . . . .	13
1.5. torre4.png . . . . .	13
1.6. torre1-2.png . . . . .	13
1.7. torre2-2.png . . . . .	13
1.8. torre3-2.png . . . . .	14
1.9. torre4-2.png . . . . .	14
1.10. torre1-3.png . . . . .	14
1.11. torre2-3.png . . . . .	14
1.12. torre3-3.png . . . . .	14
1.13. torre4-3.png . . . . .	14



# 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, la implementación de un algoritmo recursivo consiste en ejecutar repetidamente el mismo método. Los principales criterios a la hora de elegir entre programar algo iterativamente o recursivamente son: el rendimiento y la simpleza del código generado. Supongamos que debemos resolver el problema de sumar los primeros  $n$  números, dos algoritmos que solucionan este problema son los siguientes:

Iterativo

**Listing 1.1:** sumaIterativa.cpp

```
1  int sumaIterativa(int n){
2      int resultado = 0;
3      for(int i=1; i<=n; i++){
4          resultado += i;
5      }
6      return resultado;
7  }
```

Recursivo

**Listing 1.2:** sumaRecursiva.cpp

```
1  int sumaRecursiva(int n){
2      //Caso base
3      if(n==1){
4          return 1;
5      }else{
6          return n + sumaRecursiva(n-1);
7      }
8  }
```

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:  $0+1+2+3+\dots+n$ , pero en el recursivo sumamos desde  $n$  hasta 0:  $n+(n-1)+(n-2)\dots+0$ . Si quisiéramos que tuvieran un comportamiento más similar podríamos programar el algoritmo recursivo de la siguiente manera:

**Listing 1.3:** sumaRecursiva2.cpp

```

1  int sumaRecursiva(int actual, int n){
2      //Caso base
3      if(actual==n){
4          return actual;
5      }else{
6          return actual + sumaRecursiva(actual+1,n);
7      }
8  }

```

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  $n * \text{factorial de } (n-1)$ , y la solución de factorial de  $(n-1)$  es  $(n-1) * \text{factorial de } (n-2)$  y asi constantemente.

Por ejemplo factorial de 5 es

$$\begin{aligned}
 f(5) &= 5 * f(4) \\
 f(4) &= 4 * f(3) \\
 f(3) &= 3 * f(2) \\
 f(2) &= 2 * f(1) \\
 f(1) &= 1 \\
 \text{por lo tanto} \\
 f(2) &= 2 * f(1) = 2 * 1 = 2 \\
 f(3) &= 3 * f(2) = 3 * 2 = 6 \\
 f(4) &= 4 * f(3) = 4 * 6 = 24 \\
 f(5) &= 5 * f(4) = 5 * 24 = 120
 \end{aligned}$$

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  $f(5) \rightarrow f(4) \rightarrow f(3) \rightarrow f(2) \rightarrow f(1)$

Iterativo

**Listing 1.4:** factorialIterativo.cpp

```

1  int factorial(int n){
2      if(n==0) return 1;
3      int resultado = 1;
4      for(int i=n; i>=1; i--){
5          resultado*=i;
6      }
7      return resultado;
8  }

```

Iterativo

Listing 1.5: factorialRecursivo.cpp

```

1  int factorial(int n){
2      //Caso base
3      if(n==0){
4          return 1;
5      }
6      if(n==1){
7          return 1;
8      }
9      return n * factorial(n-1);
10 }

```

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

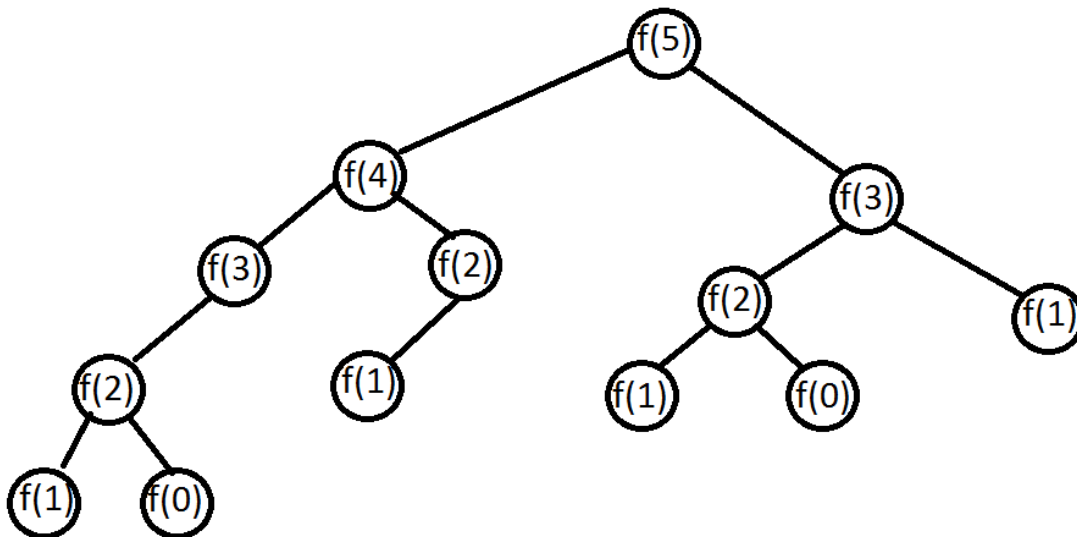


Figura 1.1: fibonacci.png

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

```

1  int fibonacci(int n){
2      if(n==0)return 0;
3      if(n==1)return 1;
4      int a = 0;
5      int b = 1;
6      int c = a+b;
7      for(int i=2;i<=n;i++){
8          c = a+b;
9          a = b;
10         b = c;
11     }
12     return c;
13 }

```

Y ahora como seria usando recursión

**Listing 1.7:** fibonacciRecursivo.cpp

```

1  int fibonacci(int n){
2      if(n==0)return 0;
3      if(n==1)return 1;
4      return fibonacci(n-1) + fibonacci(n-2);
5  }

```

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

**Listing 1.8:** euclidesIterativo.cpp

```

1  int euclides(int a,int b){
2      int temporal = a;
3      while(a>0){
4          temporal = a;
5          a = b%a;
6          b = temporal;
7      }
8      return b;
9  }

```

Recursivo

**Listing 1.9:** euclidesRecursivo.cpp

```

1  int euclides(int a,int b){
2      if(b==0)return a;
3      return euclides(b,a%b);
4  }

```

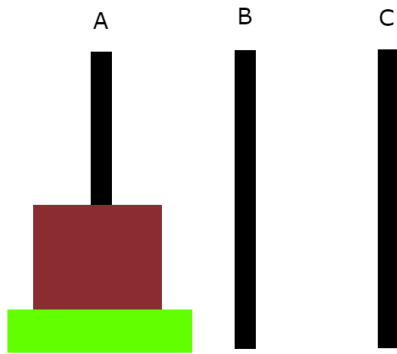
Por último mi ejemplo favorito para demostrar el potencial de la recursividad, las torres de hanoi. Si no conoces este juego, te recomiendo que primero busques en google “torres de hanoi online”, te saldrán múltiples opciones para jugarlo, es bastante simple e interesante.

En este caso no pondré 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).

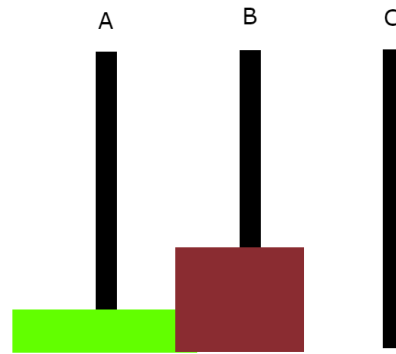
El caso base de esta solución consiste en tener únicamente dos piezas apiladas, saber donde están apiladas, hacia donde se dirigen, y el otro palo será nuestro auxiliar.

La solución al caso base es muy sencilla, únicamente debemos desplazar la ficha superior a nuestro palo auxiliar, la ficha base a nuestro palo destino y por último la ficha superior a nuestro destino, y

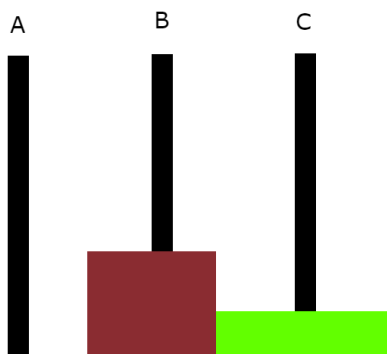
asi logramos resolver la torre de hanoi de nuestro caso base.



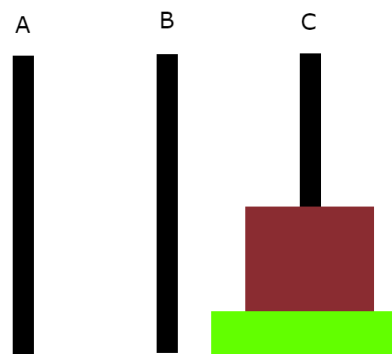
**Figura 1.2:** torre1.png



**Figura 1.3:** torre2.png

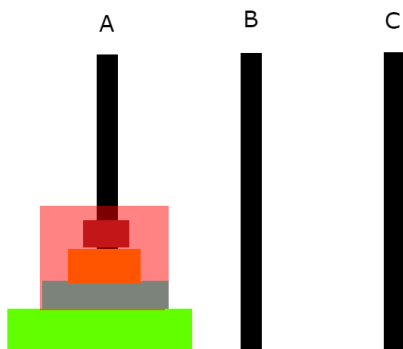


**Figura 1.4:** torre3.png

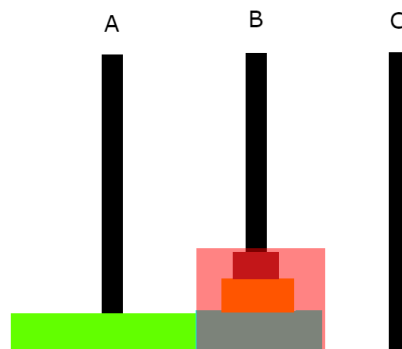


**Figura 1.5:** torre4.png

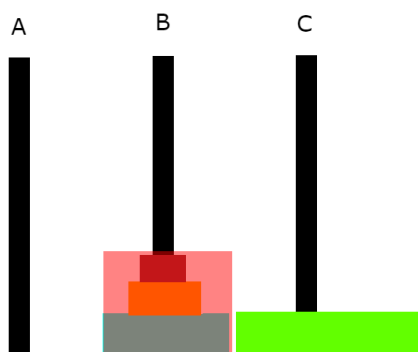
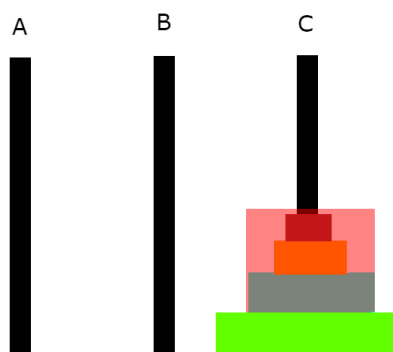
Pero que pasaria si fueran mas de dos fichas, he aqui donde viene la recursividad sucederia algo asi



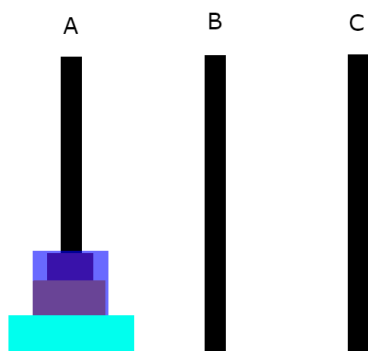
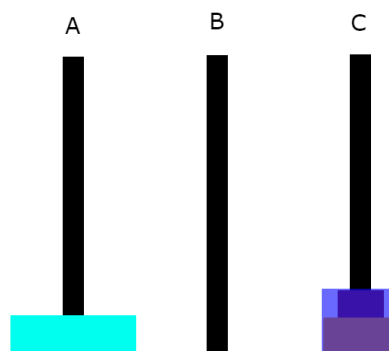
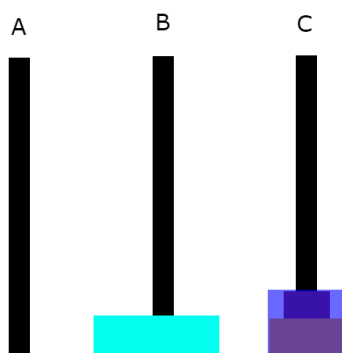
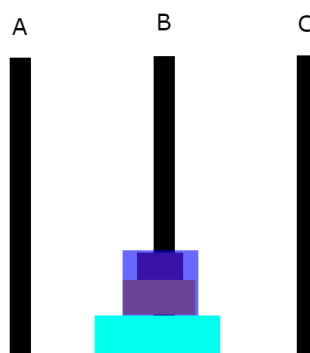
**Figura 1.6:** torre1-2.png



**Figura 1.7:** torre2-2.png

**Figura 1.8:** torre3-2.png**Figura 1.9:** torre4-2.png

Internamente la recursión de las fichas sombreadas en rojo desde “torre1-2.png” hacia “torre2-2.png” funcionarían 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 así sucesivamente (recursivamente). La solución recursiva de la torre de hanoi 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 más de una pieza, se realiza la recursión cambiando invirtiendo el palo destino y el auxiliar. En código sería así:

## Recursivo

Listing 1.10: hanoi.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 de 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. aritmetica modular

### 2.3.2. máximo común divisor y mínimo común múltiplo

#### máximo común divisor

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

#### mínimo común múltiplo

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

### 2.3.3. algoritmo de Euclides extendido

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

```

1 // store x, y, and d as global variables
2 void extendedEuclid(int a, int b) {
3     if (b == 0) { x = 1; y = 0; d = a; return; }
4     // base case
5     extendedEuclid(b, a % b);
6     // similar as the original gcd
7     int x1 = y;
8     int y1 = x - (a / b) * y;
9     x = x1;
10    y = y1;
11 }
```

### 2.3.4. 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  $x$  y  $y$  que satisfagan la ecuación, la ecuación tiene múltiples soluciones y sirven comúnmente para solucionar congruencias como  $ax + b \equiv cx + d \pmod m$  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  $y = y_0 - \frac{a}{d}n$

### 2.3.5. conjunto $z_n$

el conjunto  $z_n$  es el conjunto de elementos  $[0, 1, 2, \dots, n-1]$

### 2.3.6. propiedades de la aritmética modular

- si  $a \equiv b \pmod n$  entonces  $a + c \equiv b + c \pmod n$
- si  $a \equiv b \pmod n$  entonces  $(a - b) | m$

- $a + b \text{ mód } n = (a \text{ mód } n + b \text{ mód } n) \text{ mód } n$
- $ab \text{ mód } n = ((a \text{ mód } n)(b \text{ mód } n)) \text{ mód } n$
- si  $a \equiv b \text{ mód } n$  entonces  $ac \equiv bc \text{ mód } n$

### 2.3.7. 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  $ax \equiv 1 \text{ mód } n$  para ello se usa el algoritmo de Euclides extendido que presentamos después.

para que  $a$  sea invertible  $a$  y  $n$  tienen que ser coprimos, a continuación se muestra el algoritmo para hallar el inverso

```

1 long inverse_Zn(int a,int n){
2     extendedEuclid(a,n);
3     if(d!=1){
4         return -1;
5     }
6     else{
7         if(x<0){
8             x+=n;
9             return x;
10        }
11    }
12 }
```

## 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  $a_1 * a_2 \dots * a_n$

### 2.4.2. numero de permutaciones de $n$ elementos

el numero 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. numero 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

### formas de calcular los números combinados o coeficientes binomiales

formula:

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

formula recursiva:

$$\text{casos bases } c(n, n) = c(n, 0) = 1$$

$$\text{de mas casos } c(n, m) = c(n-1, k) + c(n-1, k-1)$$

## 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

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

### 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. 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. formulas de geometría

$$\blacksquare \frac{a}{\sin(A)} = \frac{b}{\sin(B)} = \frac{c}{\sin(C)}$$

### 3.2. estructuras geométricas

#### 3.2.1. puntos

##### punto de enteros

```
1 // struct point_i { int x, y; };
2 // basic raw form, minimalist mode
3 struct point_i { int x, y;
4 // whenever possible, work with point_i
5 point_i() { x = y = 0; }
6 // default constructor
7 point_i(int _x, int _y) : x(_x), y(_y) {} };
8 // user-defined
```

##### punto de reales

```
1 struct point { double x, y;
2 // only used if more precision is needed
3 point() { x = y = 0.0; }
4 // default constructor
5 point(double _x, double _y) : x(_x), y(_y) {} };
6 // user-defined
```

##### ordenamiento de puntos

```
1 struct point { double x, y;
2 point() { x = y = 0.0; }
3 point(double _x, double _y) : x(_x), y(_y) {}
4 bool operator < (point other) const { // override less than operator
5 if (fabs(x - other.x) > EPS)
6 // useful for sorting
7 return x < other.x;
8 // first criteria , by x-coordinate
9 return y < other.y; } };
```

```

10 // second criteria, by y-coordinate
11 // in int main(), assuming we already have a populated vector<point> P
12 sort(P.begin(), P.end());
13 // comparison operator is defined above

```

### saber si dos puntos son iguales

```

1 struct point { double x, y;
2 point() { x = y = 0.0; }
3 point(double _x, double _y) : x(_x), y(_y) {}
4 // use EPS (1e-9) when testing equality of two floating points
5 bool operator == (point other) const {
6 return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

```

### distancia euclídea entre 2 puntos

```

1 double dist(point p1, point p2) {
2 // Euclidean distance
3 // hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
4 return hypot(p1.x - p2.x, p1.y - p2.y); }
5 // return double

```

### rotar un punto con respecto al origen

```

1 // rotate p by theta degrees CCW w.r.t origin (0, 0)
2 point rotate(point p, double theta) {
3 double rad = DEG_to_RAD(theta);
4 // multiply theta with PI / 180.0
5 return point(p.x * cos(rad) - p.y * sin(rad),
6 p.x * sin(rad) + p.y * cos(rad)); }

```

## 3.2.2. líneas

```

1 struct line { double a, b, c; };
2 // a way to represent a line

```

### hallar una recta con 2 puntos

```

1 // the answer is stored in the third parameter (pass by reference)
2 void pointsToLine(point p1, point p2, line &l) {
3 if (fabs(p1.x - p2.x) < EPS) {
4 // vertical line is fine
5 l.a = 1.0;
6 l.b = 0.0;
7 l.c = -p1.x;
8 // default values
9 } else {
10 l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
11 l.b = 1.0;

```



```
12 // IMPORTANT: we fix the value of b to 1.0
13 l.c = -(double)(l.a * p1.x) - p1.y;
14 }
```

saber si dos lineas son paralelas

```
1 bool areParallel(line l1, line l2) {
2 // check coefficients a & b
3 return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }
```

saber si 2 lineas son iguales

```
1 bool areSame(line l1, line l2) {
2 // also check coefficient c
3 return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }
```

saber si dos lineas son paralelas

```
1 bool areParallel(line l1, line l2) {
2 // check coefficients a & b
3 return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }
```

saber si 2 lineas son iguales

```
1 bool areSame(line l1, line l2) {
2 // also check coefficient c
3 return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }
```

intersección entre 2 lineas



## 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 razón 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 están 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 sería ideal. A lo que quiero llegar es que hay distintas formas de ordenar la información, y algunas sirven para mejorar el desempeño en algunas áreas 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 serán la clave para idear una solución óptima.

### 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 descripción por encima de la notación big  $O$ . Esta notación nos dice cuántas ejecuciones realizaría un algoritmo en el peor de los casos, por ejemplo si tenemos que buscar un libro dentro de la biblioteca desordenada, la complejidad sería big  $O(n)$  siendo  $n$  la cantidad de libros, ya que en el peor de los casos tendríamos 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 están organizados en línea, por ejemplo en un arreglo de izquierda a derecha.

En la mayoría de lenguajes de programación podemos distinguir entre arreglos estáticos y arreglos dinámicos, a los arreglos estáticos les definimos un tamaño y es inalterable, su ventaja es que acceder a un elemento conociendo su índice es instantáneo, por ejemplo

**Listing 4.1:** arregloEstatico.cpp

```

1  int main(){
2      string palabras[] = {"hola","adios","tres"};
3      cout<<palabras[2]<<endl;
4  }

```

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

```

1  int main(){
2      string amigos[5] = {"ana","brian","cesar","daniel","eliana"};
3      string regalos[3] = {"abrazo","reloj","perfume"};
4
5      for(int i=0;i<5;i++){
6          for(int j=0;j<3;j++){
7              cout<<"le regalo un "<<regalos[j]<<" a "<<amigos[i]<<endl;
8          }
9      }
10 }

```

Los arreglos dinamicos son iguales a los estaticos, excepto por que pueden agrandarse todo lo que quieran (que lo soporte la RAM), otra gran diferencia es que ya trae 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 índice), si no  $O(n)$
- Borrar  $O(n)$
- actualizar  $O(1)$  (cuando conocemos el índice), si no  $O(n)$

Como podemos observar, sus complejidades son muy efectivas, y por eso son muy usadas en la mayoría 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 optima. 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

```

1  int main(){
2      vector<string> amigos;
3      vector<string> regalos;
4      string amigo,regalo;
5      cout<<"ingrese todos sus amigos, uno por uno , si ya termino ingrese 0"
        <<endl;
6      while(cin>>amigo){
7          if(amigo=="0")break;
8          amigos.push_back(amigo);
9      }
10     cout<<"ingrese todos los regalos, uno por uno , si ya termino ingrese 0"
        <<endl;
11     while(cin>>regalo){
12         if(regalo=="0")break;
13         regalos.push_back(regalo);
14     }
15
16     for(int i=0;i<amigos.size();i++){
17         for(int j=0;j<regalos.size();j++){
18             cout<<"le regalo un "<<regalos[j]<<" a "<<amigos[i]<<endl;
19         }
20     }
21 }
```

Generalmente la única manera de conocer el índice del elemento que estamos buscando, es que queramos recorrer el arreglo, como lo hemos hecho en los ejemplos. Así que en la mayoría de ocasiones cuando buscamos un elemento y no estamos recorriendo el arreglo 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 rápidamente, 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 mayoría 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 búsqueda binaria, la búsqueda 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. 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:  $S$  búsquedas en un arreglo desordenado tiene una complejidad de  $O(S \times n)$  y  $S$  búsquedas en un arreglo ordenado tiene una complejidad de  $O(n \log n + S \log n)$ . Si igualamos y despejamos  $S$ , obtenemos  $S = \frac{n \log(n)}{n - \log(n)}$  por lo tanto si nuestra cantidad de búsquedas es mayor a  $S$ , vale la pena ordenar el arreglo antes de realizarlas.