Notebook de maratones

February 26, 2013

Tabla de contenidos:

- La instrucción de asignación se denota por el símbolo $\leftarrow$.
- El intercambio de dos variables (*swap* en inglés) se denota por el símbolo $\leftrightarrow$.
- Los índices de los arreglos inician en 1. Es decir si L es un arreglo, entonces su primer elemento es señalado por $L[1]$ y en consecuencia su último elemento será $L[\textsc{Length}(L)]$.

1 Temas básicos

1.1 Tamaños de tipos básicos

Tipo	Tamaño	Valor Mínimo	Valor Máximo
char	16-bit	Unicode 0	Unicode $2^{16} - 1$
byte	8-bit	-128	+127
short	16-bit	-2^{15}	$+2^{15} - 1$
		-32,768	32,767
int	32-bit	-2^{31}	$+2^{31} - 1$
		(-2,147,483,648)	(2,147,483,647)
long	64-bit	-2^{63}	$+2^{63} - 1$
		(-9,223,372,036,854,775,808)	(9,223,372,036,854,775,807)
float	32-bit	32-bit IEEE 754 floating-point numbers	
double	64-bit	64-bit IEEE 754 floating-point numbers	
boolean	1-bit	false or true	

1.2 Precisión hasta x cifras decimales

```
DecimalFormat df = new DecimalFormat();
df.setMinimumFractionDigits(x);
df.setMaximumFractionDigits(x);
DecimalFormatSymbols symbols = new DecimalFormatSymbols();
symbols.setDecimalSeparator(',');
String respuesta = df.format(numero);
System.out.println(respuesta);
```

2 Dividir y conquistar

2.1 Pasos de un algoritmo de dividir y conquistar

1. **Dividir:** Dividir el problema en instancias más pequeñas del mismo problema.
2. **Conquistar:** Resolver recursivamente cada subproblema.
3. **Combinar:** Juntar las respuestas parciales

Un algoritmo de este tipo tendrá una complejidad:

$$T(n) = aT(n/b) + f(n)$$

donde a es el número de problemas más pequeños que se resuelven recursivamente, b es la fracción del problema original que se resuelve en un suproblema y $f(n)$ es lo que cuesta combinar las respuestas.

2.2 El teorema maestro

Sean $a \geq 1$ y $b > 1$ constantes, sea $f(n)$ una función y sea $T(n)$ una función en los enteros no negativos definida por la recurrencia:

$$T(n) = aT(n/b) + f(n)$$

donde n/b puede interpretarse como $\lfloor n/b \rfloor$ o $\lceil n/b \rceil$. Entonces $T(n)$ puede ser acotada asintóticamente así:

1. Si $f(n) = O(n^{log_b a - \epsilon})$ para alguna constante $\epsilon > 0$ entonces
$T(n) = \Theta(n^{log_b a})$.
2. Si $f(n) = \Theta(n^{log_b a})$ entonces $T(n) = \Theta(n^{log_b a}\mathrm{Lg}(n))$.
3. Si $f(n) = \Omega(n^{log_b a + \epsilon})$ para alguna constante $\epsilon > 0$ y si $af(n/b) \leq cf(n)$ para alguna constante $c < 1$ y un n lo suficientemente grande, entonces $T(n) = \Theta(f(n))$.

2.3 Números Fibonacci

Aprovechando la identidad

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

y el algoritmo de dividir y conquistar para exponenciación se puede obtener el n-ésimo número fibonacci en tiempo logaritmico.

2.4 Algoritmo de Strassen (Multiplicación de matrices)

Para obtener el producto de dos matrices $AB = C$, donde A, B y C son matrices $n \times n$, asumiendo que n es una potencia exacta de 2, cada matriz se puede dividir en cuatro matrices $n/2 \times n/2$. Entonces la ecuación $AB = C$ se puede reescribir como:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} r & s \\ t & u \end{pmatrix}$$

$$\begin{aligned} P_1 &= a(f-h) & P_5 &= (a+d)(e+h) \\ P_2 &= (a+b)h & P_6 &= (b-d)(g+h) \\ P_3 &= (c+d)e & P_7 &= (a-c)(e+f) \\ P_4 &= d(g-e) \end{aligned}$$

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 & t &= P_3 + P_4 \\ s &= P_1 + P_2 & u &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

Se realizan 7 multiplicaciones de matrices $n/2 \times n/2$. La suma de matrices toma tiempo proporcional a n^2. Por lo tanto tenemos que $T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{lg 7})$

2.5 Algoritmos de ordenamiento

2.5.1 Mergesort

Input : Un arreglo $X = [x_1, x_2, \cdots, x_{n-1}, x_n]$ y tres enteros p, q y r que indican dos subarreglos de X.

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
Inicializar los arreglos L y R. Length(L)= $n_1 + 1$ y Length(R)= $n_2 + 1$
for $i \leftarrow 1$ **to** n_1 **do**
    $L[i] \leftarrow X[p + i - 1]$
for $i \leftarrow 1$ **to** n_2 **do**
    $R[i] \leftarrow X[q + i]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow p$ **to** r **do**
    if $L[i] \leq R[j]$ **then**
        $X[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    else
        $X[k] \leftarrow R[j]$
        $j \leftarrow j + 1$

Algoritmo 1: Merge(X, p, q, r), Complejidad: $O(n)$

Entrada: Un arreglo $X = [x_1, x_2, \cdots, x_{n-1}, x_n]$ y dos enteros p y r, $1 \leq p < r \leq n$ que indican un subarreglo de X.

if $p < r$ **then**
    $q \leftarrow \lfloor (p + r)/2 \rfloor$
    Mergesort(X, p, q)
    Mergesort($X, q + 1, r$)
    Merge(X, p, q, r)

Algoritmo 2: Mergesort(X, p, r), Complejidad: $O(n\text{Log}(n))$

2.5.2 Quicksort

Entrada: Un arreglo $X = [x_1, x_2, \cdots, x_{n-1}, x_n]$ y dos enteros $left$ y $right$, $1 \leq left < right \leq n$ que indican un pedazo de X.
Salida : La posición i del pivote.

$L \leftarrow$ *Número al azar en el intervalo* $[\ left, right\ ]$
/* L es la posición del pivote */
$X[\ left\ ] \leftrightarrow X[\ L\ ]$
$T \leftarrow X[\ left\ ]$
/* T es el valor del pivote */
$i \leftarrow left$
for $j \leftarrow left + 1$ **to** $right$ **do**
    if $X[\ j\ ] < T$ **then**
        $i \leftarrow i + 1$
        $X[\ i\ ] \leftrightarrow X[\ j\ ]$
$X[\ left\ ] \leftrightarrow X[\ i\ ]$
return i

Algoritmo 3: SPLIT$(X, left, right)$, Complejidad $O(n)$

Entrada: Un arreglo $X = [x_1, x_2, \cdots, x_{n-1}, x_n]$ y dos enteros $left$ y $right$, $1 \leq left < right \leq n$ que indican un subarreglo de X.

if $right - left \geq 1$ **then**
    $i \leftarrow$ SPLIT$(X, left, right)$
    QUICKSORT$(X, left, i - 1)$
    QUICKSORT$(X, i + 1, right)$

Algoritmo 4: QUICKSORT$(X, left, right)$, Complejidad: $O(n\mathrm{Log}(n))$

2.6 Encontrar en tiempo lineal el i-ésimo menor elemento de un conjunto

El algoritmo Randomized-Partition es el mismo Split que utiliza Quicksort.

Input : Un arreglo $X[1, 2, \cdots, n-1, n]$ y tres enteros p, r e i.
Salida : El i-ésimo elemento del arreglo $X[p, \cdots, r]$ cuando se encuentra ordenado.

```
if p = r then
    return X[p]
else
    q ←Randomized-Partition (X, p, r)
    k ← q − p + 1
    if i = k then
        return X[q]
    else if i < k then
        return Randomized-Select (X, p, q − 1, i)
    else
        return Randomized-Select (X, q + 1, r, i − k)
```

Algoritmo 5: Randomized-Select(X, p, r, i), Complejidad esperada: $\Theta(n)$

3 Estructuras de datos

3.1 Heaps

Para mantener el máximo (o el mínimo) de un conjunto y sacarlo rápidamente:

```java
public class Heap<T extends Comparable<T>> {

        public T[] A;

        private int heapSize;

        // Constructor
        public Heap(T[] X) {
                A = X;
                heapSize = A.length;
                for (int i = (int)Math.floor(heapSize/2); i >=0; i--)
                        maxHeapify(i);
        }

        //Indice del padre de i
        public int parent(int i){
                return (int)Math.floor(i/2);
        }

        //Indice del hijo izquierdo de i
        public int left(int i){
                return 2*i;
        }

        //Indice del hijo derecho de i
        private int right(int i){
                return 2*i+1;
        }

        private void maxHeapify(int i){
                int l = left(i);
                int r = right(i);
                int largest = l< heapSize && A[l].compareTo(A[i])>0 ? l : i;
                if( r< heapSize && A[r].compareTo(A[largest])>0)
                        largest = r;
                if( largest!=i ){
                        T temp = A[i];
                        A[i] = A[largest];
```

```
                A[largest] = temp;
                maxHeapify(largest);
            }
    }

    //Devuelve el maximo elemento del Heap y lo quita
    public T removeMax(){
            T temp = A[0];
            A[0] = A[heapSize-1];
            heapSize--;
            maxHeapify(0);
            return temp;

    }

    //Ordena A de menor a mayor
    public void heapsort(){
            for (int i = heapSize-1; i >=0; i--) {
                T temp = A[0];
                A[0] = A[i];
                A[i] = temp;
                heapSize--;
                maxHeapify(0);
            }
    }

}
```

4 Teoría de números

Número de factores de un número:

La factorización prima de un número n contiene a lo sumo $\log_2 n$ factores.

Pequeño Teorema de Fermat:

Si p es un primo y a es un número natural positivo primo relativo con p entonces $a^{p-1} \equiv 1 \mod p$.

Teorema chino del residuo:

Sean $m_1, m_2, \cdots m_n$ enteros de los cuales todos son primos relativos entre sí. Además sean $a_1, a_2, \cdots a_n$ enteros arbitrarios. Entonces el sistema:

$$\begin{aligned} x &\equiv a_1 \mod m_1 \\ x &\equiv a_2 \mod m_2 \\ &\vdots \\ x &\equiv a_n \mod m_n \end{aligned}$$

tiene una única solución módulo $m = m_1 m_2 \cdots m_n$

Función Phi de Euler:

La función $\phi(n)$ denota el número de enteros en $\{1, 2, \cdots, n\}$ que son primos relativos a n.

- Si a y b son primos relativos entonces $\phi(ab) = \phi(a)\phi(b)$
- Si p es un primo entonces $\phi(p^k) = p^k - p^{k-1}$.

Número de primos hasta n:

Sea $\pi(n)$ el número de primos entre $1, 2, \cdots, n$. Entonces $\pi(n) \sim \frac{n}{\ln(n)}$

4.1 Algoritmo euclidiano

Entrada: $a, b \in \mathbb{N}$, por lo menos uno de ellos distinto de cero.
Salida : El máximo común divisor de a y b.

```
if b = 0 then
    return a
else
    return GCD(b, a (mod b))
```

Algoritmo 6: GCD(a, b), Complejidad: $O(\text{Log}(a)+\text{Log}(b))$

4.2 Algoritmo euclidiano extendido

Entrada: $a, b \in \mathbb{N}$, por lo menos uno de ellos distinto de cero.
Salida : El máximo común divisor d de a y b y los valores de x y y tales que $d = ax + by$.

if $b = 0$ **then**
  return $(a, 1, 0)$
$(d', x', y') \leftarrow$ Extended-GCD$(b, a \ (mod\ b))$
$(d, x, y) \leftarrow (d', y', x' - \lfloor a/b \rfloor) \cdot y'$
return (d, x, y)

Algoritmo 7: Extended-GCD(a, b), Complejidad: $O(\text{Log}(a)+\text{Log}(b))$

4.3 Como resolver ecuaciones lineales modulares

Entrada: $a, n \in \mathbb{N}^+$ y $b \in \mathbb{Z}$
Salida : Siendo $d = gcd(a, n)$ retorna los d distintos valores de x tales que $ax \equiv b(mod\ n)$.

$(d, x', y') \leftarrow$Extended-GCD(a, n)
if $d \mid b$ **then**
  $x_0 \leftarrow x'(b/d) mod\ n$
  for $i \leftarrow 0$ **to** $d - 1$ **do**
    print $(x_0 + i(n/d)) mod\ n$
else
  print No solutions

Algoritmo 8: Mod-eq-solv(a, b, n), Complejidad: $O(\text{Log}(n)+\text{gcd}(a, n))$

4.4 Algoritmo de exponenciación modular (Computar $b^n \mod m$)

Entrada: $b, n, m \in \mathbb{Z}$
Salida : $b^n \mod m$

Computar la representación binaria de $n = (a_{k-1}, \cdots a_1, a_0)$
$x \leftarrow 1$
$potencia \leftarrow b \mod m$
for $i \leftarrow 0$ **to** $k - 1$ **do**
  if $a_i = 1$ **then**
    $x \leftarrow (x \cdot potencia) \mod m$
  $potencia \leftarrow (potencia \cdot potencia) \mod m$
return x

Algoritmo 9: Mod-exp(b, n, m), Complejidad (bits): $O((\text{Log}^2(m))\text{Log}(n))$

4.5 Como verificar si un número es primo

Algunos hechos:

- Todos los primos mayores que 3 pueden escribirse en la forma $6k + / - 1$
- Cualquier número n puede tener solamente un factor primo mayor a $\sqrt{n}$

Entrada: Un entero positivo n
Salida : $True$ si n es primo, $False$ de lo contrario.

if $n = 1$ **then**
  return False
if $n < 4$ **then**
  return True
if $n \mod 2 = 0$ **then**
  return False
if $n < 9$ **then**
  return True
if $n \mod 3 = 0$ **then**
  return False
$i \leftarrow 5$
while $i \leq \lfloor \sqrt{n} \rfloor$ **do**
  if $n \mod i = 0$ ***or*** $n \mod (i + 2) = 0$ **then**
    return False
  $i \leftarrow i + 6$
return True

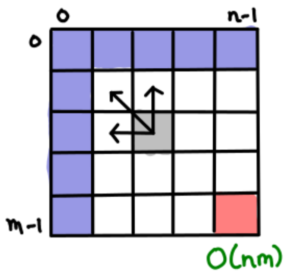
Algoritmo 10: ES-PRIMO(n), Complejidad:$O(\sqrt{n})$

4.6 Conseguir los divisores de los primeros n números

```java
static ArrayList<HashSet<Integer>> divisores(int n){
        ArrayList<HashSet<Integer>> div = new ArrayList<HashSet<Integer>>(n+1);
        div.add(0, null);
        div.add(1, new HashSet<Integer>());
        div.get(1).add(1);
        for (int i = 2; i <= n; i++) {
                int j=2;
                div.add(i, new HashSet<Integer>());
                div.get(i).add(1);
                while(j<=Math.sqrt(i)){
                        if(i%j==0){
                                div.get(i).add(j);
                                div.get(i).add(i/j);
                                HashSet<Integer> A = div.get(j);
                                HashSet<Integer> B = div.get(i/j);
                                for(Integer a:A){
                                        div.get(i).add(a);
                                        for(Integer b:B)
                                                div.get(i).add(a*b);
                                }
                                for(Integer b:B)
                                        div.get(i).add(b);
                                break;
                        }
                        j++;
                }
                div.get(i).add(i);
        }
```

```
            return div;
    }
```

5 Programación dinámica



5.1 Partición balanceada

Dado un conjunto $A_1, \cdots A_n$ de n enteros (en el rango $0 \cdots k$)se quiere particionar en dos conjuntos S_1 y S_2 tales que se minimize $|Sum(S_1) - Sum(S_2)|$. Para resolverlo se define una función $P : \{0, 1, \cdots n\} \times \{0, 1, \cdots nk\} \to \{0, 1\}$ que:

$$P(i,j) = \begin{cases} 1 & \text{Si un subconjunto de } \{A_1, A_2 \cdots A_i\} \text{ tiene una suma igual a } j \\ 0 & \text{De lo contrario} \end{cases}$$

$$P(i,j) = \begin{cases} 1 & \text{Si } P(i-1,j) = 1 \text{ o si } P(i-1, j-A_i) = 1 \\ 0 & \text{De lo contrario} \end{cases}$$

Sea $S = \sum_{i=1}^{n} A_i/2$. Entonces el valor a conseguir con la recurrencia sería:

$$Min_{i \leq S}\{S - i : P(n,i) = 1\}$$

La suma de cada conjunto sería entonces $Sum(S_1) = i$ y $Sum(S_2) = 2S - i$. Como la diferencia es $2S - 2i = 2(S - i)$ el valor objetivo es $2Min_{i \leq S}\{S - i : P(n,i) = 1\}$. La complejidad es $O(n^2k)$.

5.2 Subsecuencia contigua máxima

Dado un arreglo de números reales $A[1], A[2], \cdots, A[n]$, determinar una subsecuencia $A[i], \cdots, A[j]$ para la cual la suma de sus elementos es máximizada.

$M(i)$ = Suma máxima de las subsecuencias que terminan en $A[i]$.

$$M(i) = Max \begin{cases} M(i-1) + A[i] & \text{(Extiende la subsecuencia anterior)} \\ A[i] & \text{(Empieza una nueva subsecuencia)} \end{cases}$$

El valor máximo es el mayor entre $M(1) \cdots M(n)$ ya que no sabemos en que casilla va a terminar la subsecuencia contigua máxima. La complejidad derá $O(n)$.

5.3 Problema del mochilero (The integer (0/1) knapsack problem, duplicate items forbidden)

Dados n items, cada uno con tamaño entero S_i y un valor también entero V_i queremos conocer la forma de llenar (no necesariamente de forma exacta) una mochila de capacidad entera C con un subconjunto de los items de forma que se maximize la sumatoria de sus valores.

$M(i,j)$ = Valor óptimo para llenar exactamente un mochilero de capacidad j con algún subconjunto de elementos $1 \cdots i$.

$$M(i,j) = Max \begin{cases} M(i-1,j) & \text{(Sin utilizar el } i\text{-ésimo ítem)} \\ M(i-1, j-S_i) + V_i & \text{(Utilizando el } i\text{-ésimo ítem)} \end{cases}$$

El valor óptimo objetivo es $Max(M(n,j))$ evaluado sobre todos los js $(1, 2 \cdots C)$ ya que no sabemos cual va a ser la capacidad exacta de la mochila óptima. El algoritmo tendrá una complejidad de $O(nC)$.

5.4 Mínima distancia de edición

Dadas dos cadenas $A[1, \cdots n]$ y $B[1, \cdots m]$ queremos saber el mínimo número de cambios necesarios para transformar A en B. Los cambios que podemos hacer son: insertar un caracter, eliminar un caracter o reemplazar un caracter, cada uno con un costo c_i, c_e y c_r respectivamente.

$$M(i,j) = Min \begin{cases} c_e + M(i-1,j) \\ c_i + M(i,j-1) \\ \begin{cases} M(i-1,j-1) & \text{si } A[i] = B[j] \\ M(i-1,j-1) + c_r & \text{si } A[i] \neq B[j] \end{cases} \end{cases}$$

$$M(0,0) = 0$$

$$M(i,0) = i \cdot c_e$$

$$M(0,j) = j \cdot c_i$$

La solución óptima es $M(n,m)$ y el algoritmo tiene una complejidad de $O(nm)$.

5.5 Largest common subsequence (LCS)

En el problema de la subsecuencia común más larga nos dan dos secuencias $X = \langle x_1, x_2, \cdots, x_m \rangle$ y $Y = \langle y_1, y_2, \cdots, y_n \rangle$ y deseamos encontrar una subsecuencia común de longitud máxima de X y Y.

Dada una secuencia $X = \langle x_1, x_2, \cdots, x_m \rangle$ definimos el i-esimo prefijo de X, para $i = 0, 1, \cdots, m$, como

$X_i = \langle x_1, x_2, \cdots, x_i \rangle$.

Teorema: Subestructura óptima de una LCS

Sean $X = \langle x_1, x_2, \cdots, x_m \rangle$ y $Y = \langle y_1, y_2, \cdots, y_n \rangle$ secuencias y sea $Z = \langle z_1, z_2, \cdots, z_k \rangle$ cualquier LCS de X y Y:

1. Si $x_m = y_n$, entonces $z_k = x_m = y_n$ y Z_{k-1} es una LCS de X_{m-1} y Y_{n-1}.
2. Si $x_m \neq y_n$, entonces $z_k \neq x_m$ implica que Z es una LCS de X_{m-1} y Y.
3. Si $x_m \neq y_n$, entonces $z_k \neq y_n$ implica que Z es una LCS de X_m y Y_{n-1}.

Con esto se puede definir una solución recursiva. Sea $c[i,j]$ la longitud de una LCS de X_i y Y_j. Entonces:

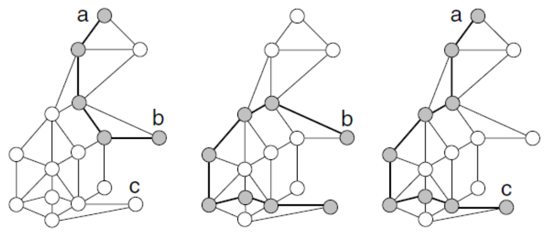
$$c[i,j] = \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0 \\ c[i-1, j-1] + 1 & \text{si } i, j > 0 \text{ y } x_i = y_j \\ Max \begin{cases} c[i, j-1] \\ c[i-1, j] \end{cases} & \text{si } i, j > 0 \text{ y } x_i \neq y_j \end{cases}$$

Entrada: Dos secuencias $X = \langle x_1, x_2, \cdots, x_m \rangle$ y $Y = \langle y_1, y_2, \cdots, y_n \rangle$.
Salida : Dos matrices c (de costos)y b (que permite reconstruir la LCS).

$m \leftarrow length[X]$
$n \leftarrow length[Y]$
Inicializar una matriz de enteros $c[m,n]$ y una matriz de símbolos $b[m,n]$
for $i \leftarrow 1$**to** m **do**
    $c[i,0] \leftarrow 0$
for $i \leftarrow 1$**to** n **do**
    $c[0,i] \leftarrow 0$
for $i \leftarrow 1$**to** m **do**
    for $j \leftarrow 1$**to** n **do**
        if $x_i = x_j$ **then**
            $c[i,j] \leftarrow c[i-1, j-1] + 1$
            $b[i,j] \leftarrow$ "$\nwarrow$"
        else if $c[i-1, j] \geq c[i, j-1]$ **then**
            $c[i,j] \leftarrow c[i-1, j]$
            $b[i,j] \leftarrow$ "$\uparrow$"
        else
            $c[i,j] \leftarrow c[i, j-1]$
            $b[i,j] \leftarrow$ "$\downarrow$"
return b y c

Algoritmo 11: LCS(X,Y), Complejidad:$O(nm)$

6 Grafos



6.1 Hechos y teoremas:

Camino euleriano: Es un camino que pasa por todos los arcos exactamente una vez.
Ciclo hamiltoniano: Es un ciclo que contiene todos los nodos del grafo.
Criterio de Euler:

- Si un grafo conexo tiene mas de dos nodos con grado impar entonces no tiene un camino euleriano.
- Si un grafo conexo tiene exactamente dos nodos con grado impar entonces tiene un camino euleriano. Todo camino euleriano debe comenzar en uno de esos nodos y terminar en el otro.
- Si un grafo conexo no tiene nodos con grado impar entonces tiene un camino euleriano. Todo camino euleriano será cerrado.

Como consecuencia se tiene que un grafo tiene un camino euleriano cerrado si y solo si todos los nodos tienen grado par.
Matriz laplaciana: Para un grafo G de n vertices su matriz laplaciana $L := (l_{i,j})_{n\times n}$ se define como:

$$l_{i,j} = \begin{cases} deg(v_i) & \text{Si } i = j \\ -1 & \text{Si } i \neq j \text{ y } v_i \text{ es adyacente a } v_j \\ 0 & \text{De lo contrario} \end{cases}$$

Teorema de Cayley: El número de árboles *etiquetados* de n nodos es n^{n-2}.

Teorema de Kirchhoff: El número de árboles de expansión de un grafo (etiquetado) es igual al valor absoluto de cualquier cofactor de la matriz laplaciana del grafo.

6.2 Breadth-first Search

Breadth-First Search (BFS) es una estrategia para recorrer todos los vertices de un grafo. Más especificamente el algoritmo BFS sirve para encontrar un vertice (o arco) que satisfaga una propiedad P o para contar cuantos la satisfacen.

Entrada: Un grafo conexo $G = (V, E)$ (y opcionalmente un nodo $v_0 \in V$ desde el que comenzar). Una propiedad P a probar.
Salida : El número de vertices que satisfacen la propiedad P.

```
Inicializar una cola Q de vertices que no han sido visitados, que en principio contenga al vertice raiz v_0.
Crear una lista T,inicialmente vacia, de vertices ya visitados.
cont ← 0
for each w ∈ Q do
    if P(w) then
        cont ← cont + 1
    Agregar a Q todos los vecinos de w que no esten en T.
    Quitar de Q todos los vertices w que han sido visitados.
    Agregar w a T.
    if T = V then
        return cont
```

Algoritmo 12: BFS(G), Complejidad: $O(|V| + |E|)$

6.3 Depth-first Search

Entrada: Un grafo conexo $G = (V, E)$ (y opcionalmente un nodo $v_0 \in V$ desde el que comenzar).
Salida : El número de vertices que satisfacen la propiedad P.

```
Inicializa un stack S
Push(S, v_0)
while S no este vacía do
    v ← Pop(S)
    Colorear v
    for cada vecino u de v do
        if u no esta coloreado then
            Push(S, u)
```

Algoritmo 13: DFS(G)

6.4 Algoritmo de Prim

Un *spanning tree* o *árbol de expansión* de un grafo G es un árbol T que contiene todo vertice de G. Si un grafo tiene costos su *minimum spanning tree* (MST) es un árbol de expansión que tiene un peso total de arcos menor o igual al de cualquier otro posible árbol.

Entrada: Un grafo $G = (V, E)$ con costos.
Salida : Un MST T de G.

```
Inicializar el árbol T = (V_T, E_T). V_T = {v_0} donde v_0 es un vertice arbitrario. E_T = ∅.
while V_T ≠ V do
    Elegir el arco (u, v) de peso mínimo tal que u este en V_T pero no v.
    Agregar v a V_T y (u, v) a E_T.
```

Algoritmo 14: Prim(G)

6.5 Algoritmo de Dijkstra

Para encontrar la distancia minima para ir desde un nodo s a un nodo d. Dependiendo de que estructura se utilize para representar a Q su complejidad varia. Por ejemplo si se hace con un heap

```java
        // Cada nodo tiene un atributo de distancia que dice cual
        // es la menor distancia hasta ese nodo y un indice que
        // lo ubica dentro del arreglo del heap
        public static long dijkstra(nodo[] nodos, int s, int d){
                int N = nodos.length;
                for (int i = 0; i < N; i++)
                        nodos[i].dist = Long.MAX_VALUE;
                nodos[s].dist = 0L;
                MinHeap Q = new MinHeap(nodos);
                while(Q.heapSize>0){
                        nodo min = Q.removeMin();
                        // Busca el nodo con menor distancia
                        // (y que no ha sido visitado)
                        for (arco arc: min.vec) {
                                // Entre sus vecinos actualiza
                                // la distancia en caso de ser necesario
                                long alt = arc.longit + min.dist;
                                nodo vecino = nodos[arc.nod.id];
                                if(alt<vecino.dist)
                                        Q.changeKey(vecino.ind, alt);
                        }
                }
                return nodos[d].dist;
        }
```

6.6 Ordenamiento topológico:

Entrada: Un grafo dirigido sin ciclos $G = (V, E)$.
Salida : Un ordenamiento topológico de G

Inicializar una cola Q
Inicializar un arreglo A (el ordenamiento)
Computar un arreglo $id[1, 2, \cdots, |V|]$ que indica el *in degree* (grado de entrada) de cada vértice.
Agregar a la cola los $v \in V$ tales que $id[v] = 0$
while Q *no vacía* **do**
    $u \leftarrow Q.\textsc{Remove}\ ()$
    $A.\textsc{Add}(u)$
    for each $v \in \textsc{adj}(u)$ **do**
        $id[v] = id[v] - 1$
        if $id[v] = 0$ **then**
            $Q.\textsc{Add}(v)$
return A

Algoritmo 15: $\textsc{Topological-Sort}(G)$, Complejidad:$O(|V| + |E|)$

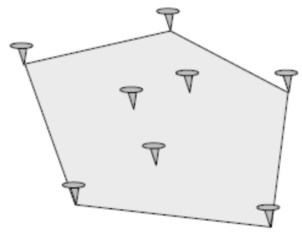
6.7 Componentes fuertemente conexas:

```java
import java.util.ArrayList;
import java.util.Stack;

import p4.Grafo;
import p4.Nodo;

public class ComputarSCC {

        public Grafo G;

        public Nodo s; //Nodo de origen

        public int t; //Tiempo actual

        public boolean[] explorados;
```

```java
public Nodo[] ordenamiento; //Ordenamiento de los nodos

public ArrayList<Integer> tamaniosSCCs; //Tamanio de cada una de las SCC

public int SCC; //Tamanio de la SCC explorada en el momento

public ComputarSCC(Grafo g) {
        super();
        G = g;
        t = 0;
}


public void SCC(){
        explorados = new boolean[ G.nodos.length ];
        ordenamiento = new Nodo[ G.nodos.length ];

        DFS_loop_inv();

        explorados = new boolean[ G.nodos.length ];
        tamaniosSCCs = new ArrayList<Integer>();

        DFS_loop();
}

public void DFS_loop_inv(){
        for(Nodo n:G.nodos)
                if(!explorados[n.id])
                        DFS_inv(n);
}

public void DFS_inv(Nodo i){
        Stack<Nodo> stack = new Stack<Nodo>();
        Stack<Nodo> stack2 = new Stack<Nodo>();
        stack.push(i);
        while(stack.size()>0){
                Nodo j = stack.pop();
                if(!explorados[j.id]){
                        stack2.push(j);
                        explorados[j.id] = true;
                        for(Nodo k:j.vecinosInversos)
                                if(!explorados[k.id])
                                        stack.push(k);
                }

        }
        while(!stack2.isEmpty()){
                        ordenamiento[t] = stack2.pop();
                        t++;
        }
}

public void DFS_loop(){
        for (int i = ordenamiento.length-1; i >= 0; i--) {
                Nodo n = ordenamiento[i];
                if(!explorados[n.id])
                        DFS(n);
                if(SCC!=0){
                        tamaniosSCCs.add( SCC );
                        SCC=0;
                }
        }
        if(SCC!=0)
                tamaniosSCCs.add( SCC );
}

public void DFS(Nodo i){
        Stack<Nodo> stack = new Stack<Nodo>();
        stack.push(i);
        while(stack.size()>0){
```

```
                Nodo j = stack.pop();
                if(!explorados[j.id]){
                    SCC++;
                    explorados[j.id] = true;
                    for(Nodo k:j.vecinos)
                        if(!explorados[k.id])
                            stack.push(k);
                }
            }
        }

}
```

7 Geometría computacional



7.1 Hechos y teoremas útiles

En un triángulo:

Ley de Senos:

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

Ley de Cosenos:

$$a^2 = b^2 + c^2 - 2bc \cdot Cos(A)$$

Ángulo entre dos vectores:

$$Cos(\theta) = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}| \cdot |\vec{v}|}$$

Producto cruz:

La magnitud del producto cruz de dos vectores $\vec{p_1}$ y $\vec{p_2}$ es el area del paralelogramo generado por ambos puntos, el origen y el punto $\vec{p_1} + \vec{p_2}$. En $\mathbb{R}^2$ el producto cruz es:

$$\vec{p}_1 \times \vec{p}_2 = (x_1, y_1) \times (x_2, y_2) = x_1 y_2 - x_2 y_1$$

Si $\vec{p_1} \times \vec{p_2} > 0$ entonces el giro de $\vec{p_1}$ a $\vec{p_2}$ es en sentido de las manecillas del reloj. Si $\vec{p_1} \times \vec{p_2} < 0$ entonces el sentido es contrario al de las manecillas del reloj. Si es igual a cero entonces $\vec{p_1}$ y $\vec{p_2}$ son colineales.

Area y centroide:

Para un polígono simple (que no tiene segmentos de línea que se intersectan) el area y centroide son:

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)$$

$$C_x = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$C_y = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

Teorema de Pick:

Dado un polígono simple construido en una rejilla de puntos equidistantes, tales que todos los vertices del polígono son puntos en la rejilla, el teorema de Pick provee una fórmuka para calcular el area A del polígono en términos del número i de puntos en el interior del polígono y el número b de puntos en el perímetro del polígono:

$$A = i + \frac{b}{2} - 1$$

Intersección entre dos rectas dadas por cuatro puntos $P_1{=}(x_1, y_1)$, $P_2{=}(x_2, y_2)$, $P_1'{=}(x_1', y_1')$ y $P_2'{=}(x_2', y_2')$:

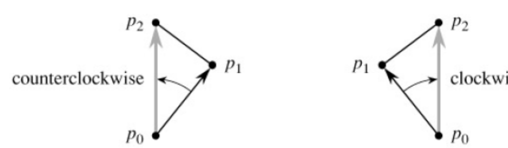
$$m := \frac{y_2 - y_1}{x_2 - x_1} \quad b := y_1 - m x_1$$

$$m' := \frac{y_2' - y_1'}{x_2' - x_1'} \quad b' := y_1' - m' x_1'$$

$$x = \frac{b' - b}{m - m'} \quad y = mx + b$$

7.2 Determinar si dos segmentos de recta consecutivos realizan un giro a izquierda o a derecha

Teniendo dos segmentos de recta $\overline{p_0 p_1}$ y $\overline{p_1 p_2}$ queremos saber si el giro en p_1 es a izquierda o a derecha. Para esto se puede utilizar el producto cruz:



Si se computa el producto cruz $(\vec{p_2} - \vec{p_0}) \times (\vec{p_1} - \vec{p_0})$ y da negativo entonces el giro es en contra de las manecillas del reloj y por lo tanto hacia la izquierda.

7.3 Distancia de un punto a un segmento

Entrada: El punto P que se quiere evaluar y los dos puntos extremos P_1 y P_2 del segmento de recta.
Salida : La menor distancia del punto al segmento de recta

$d \leftarrow$ Distancia-Entre-Puntos(P_1, P_2)
$d_1 \leftarrow$ Distancia-Entre-Puntos(P_1, P)
$d_2 \leftarrow$ Distancia-Entre-Puntos(P_2, P)
if $d + d_2 < d_1$ **or** $d + d_1 < d_2$ **then**
    return Min(d_1, d_2)
else
    return Distancia a recta(P, P_1, P_2)

Algoritmo 16: Distancia-Punto-a-Segmento(P, P_1, P_2)

7.4 Distancia de un punto a una recta

Entrada: El punto P que se quiere evaluar y dos puntos P_1 y P_2 que estan dentro de la recta. $P_1 \neq P_2$.
Salida : La menor distancia del punto a la recta.

$m \leftarrow \frac{P_{2y} - P_{1y}}{P_{2x} - P_{1x}}$
$m' \leftarrow -\frac{1}{m}$
$b \leftarrow P_{1y} - mP_{1x}$
$b' \leftarrow P_y - m'P_x$
$P' \leftarrow (\frac{b'-b}{m-m'}, mx + b)$
return Distancia-Entre-Puntos(P, P')

Algoritmo 17: Distancia-Punto-A-Recta(P, P_1, P_2)

7.5 Computar el *convex hull*

Dado un conjunto Q de puntos su *convex hull* es el polígono P mas pequeño tal que todo punto se encuentra en el borde o dentro de P.

7.5.1 Graham scan

Entrada: Un conjunto Q de puntos en $\mathbb{R}^2$. Q contiene al menos 3 puntos no colineales.
Salida : Una pila que, del fondo al tope, contiene los puntos del convex hull en sentido de las manecillas del reloj.

Sea p_0 el punto con menor coordenada en y o el de menor coordenada en x en caso de haber empate.
Sean $< p_1, p_2, \cdots, p_m >$ los puntos restantes en Q ordenados por ángulo polar en sentido contrario a las manecillas del reloj alrededor de p_0. Si hay mas de uno con el mismo ángulo dejar solamente el que esta mas lejos de p_0.
Inicializar una pila S
S.Push(p_0)
S.Push(p_1)
S.Push(p_2)
for $i = 3$ **to** m **do**
    while *el ángulo formado por los puntos* S.Next-To-Top(), S.Top() *y* p_i *tenga un giro que no sea hacia la izquierda* **do**
        S.Pop()
    S.Push(p_i)
return S

Algoritmo 18: Graham-Scan(Q), Complejidad:$O(n\text{Log}(n))$

8 *String matching*

Dado un arreglo de texto $T[1, 2, \cdots, n]$ de longitud n se desea saber si un patrón $P[1, 2, \cdots, m]$ de longitud $m \leq n$ es una subcadena de T.

8.1 Algoritmo de Rabin-Karp

Idea: interpretar cada una de las subcadenas de longitud m de T (Son aquellas $T[s+1, s+2, \cdots, s+m]$ para $s = 0, 1, \cdots, n-m$) como un número en base $d = |\Sigma|$. Sin embargo, siendo el tamaño del alfabeto muy grande los números se vuelven inmanejables por lo que se propone utilizar los números módulo un q primo. El primo q debería ser elegido de forma que $q \cdot d$ apenas quepa en una palabra del computador. Aún así pueden haber "coincidencias espurias": cuando los números son distintos definitivamente el patrón no coincide pero si son iguales no se puede afirmar que el patrón ocurra. Esto sucede debido a que $t_s \equiv p \mod q$ no implica $t_s = p$. Por eso se hace necesario verificarlo explícitamente. Los subíndices de t estan por claridad. Se pueden quitar.

```
Entrada: Dos cadenas T y P, el tamaño del alfabeto d y un primo q adecuado.
Salida  : Imprime las ocurrencias del patrón P en T.
m ← P.Length
n ← T.Length
/* Constante para calcular el siguiente t_s                    */
h ← d^(m−1) mod q
p ← 0
t_0 ← 0
/* Preprocesamiento                                            */
for i ← 1 to m do
    p ← (d · p + P[i]) mod q
    t_0 ← (d · t_0 + T[i]) mod q
/* Matching                                                    */
for s ← 0 to n − m do
    if p = t_s then
        if P[1 ··· m] = T[s + 1 ··· s + m] then
            print "El patrón ocurre en " s
    if s < n − m then
        t_{s+1} ← (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```

Algoritmo 19: Rabin-Karp-Matcher(T, P, d, q), Complejidad:$\Theta((n-m+1)m)$

9 Combinatoria

9.1 Hechos y teoremas útiles

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1

Permutaciones:

El número de permutaciones (posibles reordenamientos) de n objetos es $n!$

Subconjuntos:

El número de subconjuntos de k elementos formados a partir de uno de n elementos es $\binom{n}{k} = \frac{n!}{(n-k)!k!}$.

Subconjuntos ordenados:

El número de subconjuntos ordenados de k elementos formados a partir de uno de n elementos es $n(n-1)\cdots(n-k+1) = \frac{n!}{(n-k)!}$.

El teorema binomial:

Los coeficientes de $x^{n-k}y^k$ en la expansión de $(x+y)^n$ es el coeficiente binomial $\binom{n}{k} = \frac{n!}{(n-k)!k!}$. En otras palabras tenemos la identidad:

$$(x+y)^n = \binom{n}{0}x^n + \binom{n}{1}x^{n-1}y + \cdots + \binom{n}{n-1}xy^{n-1} + \binom{n}{n}y^n$$

Algunas propiedades de los coeficientes binomiales:

- Simetría

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

- Identidad de Pascal

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

- Subconjuntos de un conjunto

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

- Identidad de Vandermonde

$$\binom{m+n}{r} = \sum_{k=0}^{r} \binom{m}{r-k}\binom{n}{k}$$

9.2 Como generar el número de combinaciones

Consigue n combinado k en $O(n^2)$ por programación dinámica:

```java
public class ManejadorCombinaciones{

	//Donde se guardan las combinaciones/ Triangulo de Pascal
	private long [][] comb;

	//Constructor
	public ManejadorCombinaciones(int N) {
		comb = new long[N + 1][];
		comb[0] = new long[1];
		comb[0][0] = 1L;
		comb[1] = new long[2];
		comb[1][0] = 1L;
		comb[1][1] = 1L;
		for (int i = 2; i <= N ; i++) {
			comb[i] = new long[i/2 + 1];
			comb[i][0] = 1L;
			for (int j = 1; j < i/2; j++)
				comb[i][j] = comb[i - 1][j - 1] + comb[i - 1][j];
			comb[i][i/2] = i%2==0 ? 2*comb[i-1][i/2-1]: comb[i-1][i/2]+comb[i-1][i/2-1];
		}
	}

	//Retorna n combinado k
	public long darCombinacion(int n, int k) {
		if (k > n / 2)
			return comb[n][n - k];
		return comb[n][k];
	}
}
```

10 Probabilidad y estadística

10.1 Hechos y teoremas útiles

Para dos eventos A y B:

$$\begin{aligned} P[A \cup B] &= P[A] + P[B] - P[A \cap B] \\ &\leq P[A] + P[B] \end{aligned}$$

Dose eventos A y B son independientes si:

$$P[A \cap B] = P[A]P[B]$$

Probabilidad condicional: La probabilidad condicional de que un evento A suceda, dado que se sabe que un evento B sucedió, se define como:

$$P[A|B] = \frac{P[A \cap B]}{P[B]}$$

Teorema de Bayes:

$$P[A|B] = \frac{P[A]P[B|A]}{P[A]P[B|A] + P[\bar{A}]P[B|\bar{A}]}$$

11 Álgebra lineal

11.1 Resolver sistemas lineales de ecuaciones:

```java
/**
 * Realiza la descomposicion LUP de una matriz cuadrada
```

```
 */
public class LUPdescomposition {

        // n es la dimension de la matriz
        private int n;

        // A es la matriz que contiene a L y a U de la matriz que inicializa la clase
        // A[i][j] = L[i][j] si i>j
        // A[i][j] = U[i][j] si i<=j
        private double[][] A;

        // perm es el arreglo que representa la matriz de permutacion P
        // perm[i] indica que P[i][ perm[i] ]=1 y P[i][j]=0 para j!=perm[i]
        private short [] perm;

        // Inicializa los atributos de forma que quede listo para resolver sistemas de ecuaciones
        // Arroja excepcion si la matriz es singular
        public LUPdescomposition(double[][] matriz) throws Exception {
                n = matriz.length;
                perm = new short[n];

                A = new double[n][n];
                for (int i = 0; i < n; i++)
                        for (int j = 0; j < n; j++)
                                A[i][j] = matriz[i][j];

                for (short i = 0; i < perm.length; i++)
                        perm[i] = i;

                for (int k = 0; k < n; k++) {
                        double p = 0.0;
                        int kp = -1;
                        for (int i = k; i < n; i++)
                                if (Math.abs(A[i][k]) > p) {
                                        p = Math.abs(A[i][k]);
                                        kp = i;
                                }
                        if (p == 0.0)
                                throw new Exception("La matriz es singular");
                        short temp = perm[k];
                        perm[k] = perm[kp];
                        perm[kp] = temp;

                        for (int i = 0; i < n; i++) {
                                double tem = A[k][i];
                                A[k][i] = A[kp][i];
                                A[kp][i] = tem;
                        }

                        for (int i = k + 1; i < n; i++) {
                                A[i][k] /= A[k][k];
                                for (int j = k + 1; j < n; j++)
                                        A[i][j] -= A[i][k] * A[k][j];
                        }
                }
        }

        // Dado un vector de dimension n devuelve el vector solucion del sistema de ecuaciones
        public double[] LUPsolve(double [] b)
        {
                double [] x = new double[n];
                double [] y = new double[n];
                for (int i = 0; i < n; i++) {
                        double sum = 0.0;
                        for (int j = 0; j <= i-1; j++)
                                sum+=A[i][j]*y[j];
                        y[i]= b[perm[i]] - sum;
                }
                for (int i = n-1; i >=0; i--) {
                        double sum = 0.0;
                        for (int j = i+1; j < n; j++)
```

```
                sum+=A[i][j]*x[j];
            x[i]=(y[i]-sum)/A[i][i];
        }
        return x;
    }
}
```

12 Otros

12.1 Regla de Horner

Para evaluar un polinomio de la forma: $P(x) = \sum_{k=0}^{n} a_k x^k$ se aprovecha que
$P(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n)\cdots))$ y se usa la regla de Horner.

Entrada: Una secuencia de coeficientes $a_0, a_1, \cdots, a_{n-1}, a_n$ y un valor de x.
Salida : El valor del polinomio $\sum_{k=0}^{n} a_k x^k$ evaluado en x

$i \leftarrow 0$
$y \leftarrow n$
while $i \geq 0$ **do**
    $y \leftarrow a_i + x \cdot y$
    $i \leftarrow i - 1$
return y

Algoritmo 20: Horner's-Rule($\langle a_0, a_1, \cdots a_n \rangle, x$)

12.2 n-ésima permutación lexicográfica

Retorna la n-ésima permutación de una palabra L. Se hace recursivamente.

Entrada: Una palabra L (Un arreglo de carácteres) y un valor $n \in \mathbb{N}, 1 \leq n \leq$ Length(L)!.
Salida : La n-sima permutación de L.

if Length(L)= 1 **then**
    return L
$a \leftarrow$ (Length(L)$-$1)!
for $i \leftarrow 1$ **to** Length(L) **do**
    if $a \cdot i + 1 \leq n \leq a \cdot (i+1)$ **then**
        $p \leftarrow L[\ i\ ]$
        remove $L[\ i\ ]$ **from** L
        return p concatenado con $(n - ia)$-ésima permutación lexicográfica(L)

Algoritmo 21: n-ésima permutación lexicográfica(L, n),Complejidad:$O(n^2)$

Bibliografía:

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Stein Clifford, *Introduction to Algorithms*, Segunda edición. MIT Press, 2003.

[2] Herbert S. Wilf, *Algorithms and complexity*, Segunda edición. A K Peters , 2002.

[3] David Joyner, Minh Van Nguyen, Nathan Cohen, *Algorithmic Graph Theory*, Versión 0.3. Recuperado el 30 de marzo de 2010 de `http://code.google.com/p/graph-theory-algorithms-book/`