Greedy

Toman decisiones de corto alcance, basadas en información inmediatamente disponible, sin importar consecuencias futuras. Se usan generalmente para resolver problemas de optimización. Son eficientes y fáciles de implementar. No siempre son correctos.

Se dispone de un conjunto C de candidatos de los cuales se debe seleccionar un subconjunto que optimice alguna propiedad. A medida que avanza el algoritmo, se van seleccionando candidatos y se los coloca en el conjunto S de candidatos aceptados, o R de candidatos rechazados.

esSolución() determina si un conjunto de candidatos es una solución, no necesariamente optimal, del problema.
esViable() determina si un conjunto de candidatos es posible de ser extendido para formar una solución, no necesariamente optimal, del problema.
selección() devuelve el candidato más promisorio del conjunto de aquellos que todavía no han sido considerados.

Los algoritmos greedy resuelven problemas de optimizacion, a traves de un ciclo donde se seleccionan o descartan elementos con estrategias sencillas. No siempre funcionan, es necesario asegurarse que la estrategia elegida sea correcta para el problema en cuestión. Si funcionan, el tiempo de ejecución es del orden de la cantidad de elementos por el tiempo de selección y chequeo de viabilidad, lo que los hacen algoritmos muy eficientes.

```
C ::= conjunto de candidatos; S ::= {}
WHILE (C != {} and ! esSolución(S))
    x ::= selección(C); C ::= C - {x}
    IF esViable(S + {x})
        S ::= S + {x}
IF esSolución(S)
    RETURN S
ELSE
    RETURN "No encontré soluciones"
```

Mochila

N objetos con peso w_i y valor v_i. Se quiere maximizar el valor con un peso restringido.

```
for i=1 to n
    x[i] = 0
peso = 0
while peso < W  //ciclo greedy
    i = selección()
    if (peso + w[i] < W)
        x[i] = 1
        peso += w[i]
    else
        x[i] = (W-peso)/w[i]
        peso = W
return X
```

La mejor selección es mayor valor por unidad de peso v_i/w_i. Siempre se encuentra la solución optimal

Sea X=(x1,x2,...,xn) la solución del algoritmo y Y=(y1,y2,...,yn) cualquier otra solución, se prueba que valor(X) - valor(Y) >=0. Luego X es solución optimal.

T(n) = O(nlogn)
heapsort: se ordenan los n elementos antes del ciclo greedy y la selección en cada iteración se hace en O(1).
heap: construyo heap en O(n) y por cada selección tengo que ordenar con heapify Θ(logn) y el ciclo me queda Θ(nlogn).

Más eficiente con heap. Solo ordeno los elementos necesarios, no todos.
S(n)= depende. Si creo estructura auxiliar es O(n). valorPeso: arreglo de floats de longitud **n.** **(**valor/peso,índice). Podría trabajar sobre los arreglos de entrada y salida en O(1) y el orden está dado por el espacio que ocupe el ordenamiento. El método heapSort(vp) reordena los elementos de estos arreglos en simultáneo sin usar estructuras auxiliares, tiene S(logn).

Usos: cargar camiones, aviones, cargas en gral maximizar valor/peso/volumen

Scheduling de procesos

Servidor con **n** clientes para servir con tiempos de servicio $\mathbf{t_i}$. Se quiere encontrar una secuencia de atencion que minimice el tiempo de espera de los clientes.

Algoritmo: se ordenan los procesos por orden creciente de tiempo de servicio, y se implementa el ciclo greedy. como el cuerpo del ciclo greedy es de Θ(1), y el ciclo no se repite más de n veces, el tiempo del algoritmo en general estará dominado por el tiempo del ordenamiento: Θ(n log n).

T(n) = Θ(nlogn) dado por el ordenamiento, el ciclo greedy en sí es orden n.

S(n) = dado por el ordenamiento. **heapSort() = O(logn)** por árbol de recursión. No usa estruct aux.

Correctitud: Supongamos que P no ordena los clientes en orden creciente de tiempo de servicio. Entonces podemos encontrar dos programas **a** y **b** tal que el cliente **a** es atendido antes que el cliente **b** aunque necesite más tiempo que b. Si intercambiamos las posiciones de estos dos clientes, obtenemos un nuevo orden de servicio P' donde se reducen los tiempos de espera. Así, podemos mejorar cualquier scheduling en el que un cliente sea atendido antes que otro que requiera menos servicio.
Los únicos schedulings que quedan son aquellos donde se colocan los clientes en orden creciente por tiempo de servicio. Todos estos schedulings son equivalentes y por lo tanto optimales. No se pueden mejorar.

Existen numerosas variantes de este problema (con más de un procesador, con límites a la espera de los procesos, con ganancia por la ejecución del proceso antes del límite, etc.

Usos: un procesador, un cajero en un banco, un surtidor de nafta.

Codificacion de Huffman

Se quiere minimizar el espacio que ocupa un texto cambiando la codificación de cada letra. Se utiliza codificación variable de caracteres y se asignan códigos cortos a caracteres frecuentes y códigos más largos a caracteres poco frecuentes.
Toda codificación optimal es representada por un árbol completo, donde cada nodo interno tiene exactamente dos hijos. Si hay C caracteres, se necesita un árbol binario de |C| hojas y |C| −1 nodos internos. El camino de la raíz a la hoja nos da el código de dicho caracter.

```
n = size(C)
Q.construirHeap(C)
for i=1 to n
    z = nuevo Nodo
    z.left = x = Q.min()
    z.right = y = Q.min()
    z.freq = x.freq + y.freq
    Q.insertar(z)
return Q //raiz del arbol
```

T(n) = O(nlogn) dado por el for. T(min) = T(insertar) = O(logn)

S(n) = O(n) si creo heap aux. **O(logn)** = O(heapify) si uso el mismo arreglo.

Lema 3: sea C un alfabeto, cada c perteneciente a C, con frecuencia c.freq. Sean x, y los caracteres con menor frecuencia de C. C tiene una codificación prefija optimal donde x e y son los hermanos de máxima profundidad. Demostración: sea T una cod optimal, y a, b los hermanos de menor frecuencia. Sup a.freq<=b.freq y x.freq<=y.freq. Entonces tenemos que x.freq<=a.freq y y.freq<=b.freq. Construyo T', cambiando a por x en T, y T'' cambiando b por y en T'. Se tendrá que B(T)>=B(T')>=B(T''). Como T es optimal, T'' lo será también.. Sea C' el alfabeto basado en C, que se obtiene de reemplazar x e y en C por z, tal que z.freq=x.freq+y.freq.
Lema 4: sea T' una codificación optimal de C', se puede obtener una codificación optimal de T reemplazando Z por un nodo interno con hijos x e y. Demostración: supongo T no optimal, existe T1!=T tq T1 es optimal. Aplico el lema 3 a T1, obteniendo T'', que es una codificación mejor que T'. Esto contradice que T' es optimal, y por absurdo T es optimal.
Correctitud: se prueba por inducción en las iteraciones aplicando el lema 4, y resulta en una codificación optimal.

Usos: Transmisión de texto, formatos de compresión PKZIP, GZIP.

Problema del viajante

Se tienen **n** ciudades y las distancias entre ellas en una matriz. Se quiere partir de una de ellas y visitar exactamente una vez cada ciudad, regresando al punto de partida al final, y recorriendo la menor distancia posible.
Algoritmo: Se elige en cada paso la ciudad no visitada más próxima. **NO ES CORRECTO**.
ninguna estrategia de selección directa genera un algoritmo correcto en todos los casos. para cualquier estrategia se puede encontrar ejemplos de grafos cuya solución greedy es arbitrariamente muy mala

T(n) = O(n^2).
S(n) = O(n) dado por la forma de marcar los visitados.

Usos: Minimizar recorridos

Dividir y conquistar

Consiste en

1. descomponer la instancia del problema a resolver en un conjunto de instancias más pequeñas del mismo problema.
2. resolver independientemente cada una de estas subinstancias. No se guardan resultados de instancias previamente calculadas, como en PD.
3. combinar estas soluciones en una solución a la instancia original.

Para que DYC sea eficiente las subinstancias deben ser todas de aproximadamente el mismo tamaño y se debe elegir bien el umbral. Cuando el tamaño de mi subinstancia es <= al umbral, corto y no divido más, uso el algoritmo básico.
Correctitud: Se supone la correctitud del algoritmo básico y se prueba por inducción sobre el tamaño de la instancia que la solución obtenida es correcta suponiendo la correctitud de las instancias más chicas.

Resolución de recurrencias

Teorema maestro

Sean $a \geq 1$, $b > 1$ constantes, $f(n)$ una función y $T(n)$ una recurrencia definida sobre los enteros no negativos de la forma $T(n) = aT(n/b) + f(n)$, donde n/b puede interpretarse como $\lfloor n/b \rfloor$ o $\lceil n/b \rceil$ Entonces valen:

1. *si $f(n) \in O(n^{\log_b a - \varepsilon})$ para algún $\varepsilon > 0$ entonces $T(n) \in \Theta(n^{\log_b a})$.*
2. *si $f(n) \in \Theta(n^{\log_b a})$ entonces $T(n) \in \Theta(n^{\log_b a} \lg n)$.*
3. *si $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ para algún $\varepsilon > 0$, y satisface $af(n/b) \leq cf(n)$ para alguna constante $c < 1$, entonces $T(n) \in \Theta(f(n))$.*

Ecuación característica

se aplica a ciertas recurrencias lineales con coeficientes constantes de la forma:
$T(n) = a_1T(n-1) + a_2T(n-2) + \cdots + a_kT(n-k) + b^n p(n)$ donde a_i ,$1 \leq i \leq k$,b son constantes y $p(n)$ es un polinomio en n de grado s.

para resolver la recurrencia
$T(n) = a_1 T(n-1) + a_2 T(n-2) + \cdots + a_k T(n-k) + b^n p(n)$:

1. Encontrar las raíces no nulas de la ecuación característica:
   $$(x^k - a_1 x^{k-1} - a_2 x^{k-2} - \cdots - a_k)(x-b)^{s+1} = 0$$
   Raíces: r_i ,$1 \leq i \leq l \leq k$, cada una con multiplicidad m_i
2. las soluciones son de la forma de combinaciones lineales de estas raíces de acuerdo a su multiplicidad
   $$T(n) = \sum_{i=1}^{l} \sum_{j=1}^{m_i} c_{ij} n^{j-1} r_i^n$$
3. si se necesita, se encuentran valores para las constantes c_{ij},$1 \leq i \leq l$,$0 \leq j \leq m_i - 1$ y d_i ,$0 \leq i \leq s-1$ según la recurrencia original y las condiciones iniciales (valores de la recurrencia para n = 0,1,...)

Cambio de variable

- por ejemplo para la recurrencia

$$T(n) = \begin{cases} a & \text{si n=1} \\ 2T(n/2) + n\log_2 n & \text{sino} \end{cases}$$

no se puede ninguno de los dos métodos anteriores

- se define una nueva recurrencia $S(i) = T(2^i)$, con el objetivo de llevarla a una forma en la que se pueda resolver siguiendo algún método anterior
- el caso general queda

$$S(i) = T(2^i) = 2T(2^i/2) + 2^i i = 2T(2^{i-1}) + i2^i = 2S(i-1) + i2^i$$

con $b = 2$ y $p(i) = i$ de grado 1

Multiplicación de enteros

Se quiere multiplicar dos enteros **a** y **b**, de **n** y **m** dígitos cada uno, cantidades que no son posibles de representar directamente por el HW de la máquina. Es fácil implementar una estructura de datos para estos enteros grandes, que soporte suma y resta de Θ(n+m) y productos y divisiones por la base de Θ(n+m).
El producto tendria Θ(nm), queremos mejorar ese caso. Suponemos que |n| = |m|
Separamos a y b en dos partes. a= $x.10^{n/2}$+y, b = $w.10^{n/2}$+z.
Se tienen productos más chicos, a*b = (x*10^(n/2) +y)*(w*10^(n/2) +z) = (xw)*10^(2*n/2)+ (xz+wy)*(10^n/2) +yz.
Si aplicamos DyC recursivamente nos queda que:

$$t_{DyC}(n) = \begin{cases} \Theta(n^2) & \text{si } n \leq n_0 \\ 4t_{DyC}(\lceil n/2 \rceil) + \Theta(n) & \text{sino} \end{cases}$$

T(n) = $O(n^2)$ teo maestro caso 1. No es bueno.

Se puede obtener una ecuación equivalente y se tiene entonces **3 productos**, dos productos de n/2 dígitos, un producto de a lo sumo n/2+1 dígitos, más sumas, restas y productos de potencias de la base.

Se aplica recursivamente a los productos más pequeños y se tiene la siguiente recurrencia:

$$T_{DyC}(n) = \begin{cases} \Theta(n^2) & \text{si } n \text{ es pequeño} \\ T_{DyC}(\lfloor n/2 \rfloor) + T_{DyC}(\lceil n/2 \rceil) + \\ \quad + T_{DyC}(\lfloor n/2 \rfloor + 1) + \Theta(n) & \text{sino} \end{cases}$$

Usando teorema maestro caso 1, nos queda

$$T_{DyC}(n) \in O(n^{\log 3} | n = 2^k)$$

como T(n) es eventualmente no decreciente y $n^{\log 3}$ es de crecimiento suave, entonces:
T(n) = O($n^{\log 3}$), aplicando la regla de las funciones de crecimiento suave.

Si m << n se parte a n en bloques de m. $\Theta(nm^{log(3/2)})$

S(n) = S(n/2) + n ⇒ n de representar los 4 num de n/2. Por teo maestro caso 3 **S(n) = Θ(n)**
S(n) = S(n/2) + C ⇒ C de representar los 4 num con indices. Por teo maestro caso 2 **S(n) = Θ(logn)**

Usos: Podría usarse en la exponenciación modular o criptografía

Búsqueda binaria

Se tiene un arreglo ordenado en orden creciente y se quiere encontrar el índice i tal que T[i −1] < x ≤ T[i] . Para aplicar DYC se determina en cuál mitad del arreglo debería estar x, comparándolo con el elemento del medio y luego se busca recursivamente en esa mitad.

```
function BúsqBinaria(T[i..j],x)
    IF i=j
        RETURN i
    ELSE
        k ::= (i+j) div 2
        IF x<=T[k]
            RETURN BúsqBinaria(T[i..k],x)
        ELSE
            RETURN BúsqBinaria(T[k+1..n],x)
```

$$T(m) \leq \begin{cases} a & \text{si } m = 1 \\ b + t(\lceil m/2 \rceil) & \text{sino} \end{cases}$$

T(n) = Θ($\log_2$ n) teo maestro caso 2. para peor y mejor caso.
Se puede mejorar chequeando por la igualdad x == arr[medio], se logra O(1) en el mejor caso y se empeora el peor caso (se agregan chequeos constantes).
S(n) = O($\log_2$ n). teo maestro caso 2

Mergesort

consiste en partir el arreglo en dos mitades, ordenar cada una de las mitades por separado y hacer una mezcla de estas mitades ya ordenadas

```
Mergesort(A[1..n])
IF n es pequeño
    RETURN Inserción(A)
ELSE
    //dividir
    crear A1 y A2 subarreglos de A
    //conquistar
    B1 ::= Mergesort(A1)
    B2 ::= Mergesort(A2)
    //combinar
    Mezcla(B1, B2, B)
    RETURN B
```

La partición consiste en la creación de dos mitades del arreglo original. Hay que tener cuidado con el manejo de los parámetros, para evitar duplicar los arreglos. En este caso se pasarán los índices.

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \text{ es pequeño} \\ 2T(n \div 2) + \Theta(n) & \text{sino} \end{cases}$$

Caso 2 del teorema maestro.
T(n) = O(n logn)

Si creamos 2 subarreglos en cada llamada recursiva, el espacio queda O(n):

$$S(n) = \begin{cases} \Theta(1) & si \quad n \leq umbral \\ n + T(\frac{n}{2}) & si \quad n > umbral \end{cases}$$

Podemos aplicar el caso 3 del Teorema Maestro y concluir que **S(n) = Θ(f(n)) = Θ(n)**
Para evitar esto, trabajamos sobre el mismo arreglo pasando índices y el espacio queda **Θ(logn)** (caso 2)

Es importante partir las subinstancias en tamaños iguales. En caso extremo de T(n-1) + T(1) el T(n) se va a $O(n^2)$.(ecuación característica)

Quicksort

A diferencia de mergesort, que hace una descomposición trivial pero con una recombinación costosa, el algoritmo **quicksort** pone énfasis en la descomposición. la partición del arreglo a ordenar se realiza eligiendo un elemento (el pivote), y luego partiendo en dos subarreglos con los elementos menores o iguales al pivote, y con lo elementos mayores que el pivote. Estos nuevos arreglos son ordenados en forma recursiva, y directamente concatenados para obtener la solución al problema original. es posible obtener una implementación del pivoteo en tiempo de Θ(n), incluso realizando una sola recorrida al arreglo.

```
Quicksort(A[i..j])
    IF j-i es pequeño
        Inserción(A[i..j])
    ELSE
        piv ::= A[i]
        l = Pivotear(A[i..j],piv)
        Quicksort(A[i..l-1])
        Quicksort(A[l+1..j])

Pivotear(A[i..j],piv)
    k ::= i; l ::= j+1
    REPEAT
        k ::= k+1
    UNTIL A[k]>piv or k>j
    REPEAT
        l ::= l-1
    UNTIL A[l]<=piv
    WHILE k<l
        intercambiar A[k] y A[l]
        REPEAT
            k ::= k+1
        UNTIL A[k]>piv
        REPEAT
            l ::= l-1
        UNTIL A[l]<=piv
    return l;
```

El pivote queda en su posición, los elemento a la izquierda son menores, y los de la derecha son mayores. Entonces en cada iteracion, coloco al pivote en su posición correcta.

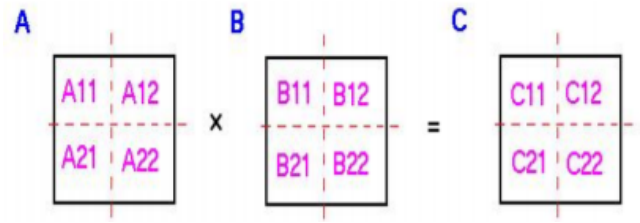
$$T(n) = \begin{cases} \Theta(1) & \text{si } n \text{ es pequeño} \\ T(n-1) + \Theta(n) & \text{sino} \end{cases}$$

$T(n) = \Theta(n^2)$ Usando la ecuación característica.
Pero en el promedio de los casos, con arreglo random, la probabilidad de las n! formas de ordenar el arreglo es la misma ⇒ **T(n) = O(nlogn)** Por inducción constructiva se encuentran los valores para c tal que T(n) ≤ cn log n.

Se puede mejorar con el pivoteo de bandera holandesa y mejor elección del pivote pero en general las constantes ocultas de este algoritmo son muy altas, y solo es conveniente su implementación en arreglos de gran cantidad de elementos.
Tiene menos constantes ocultas que mergesort.
S(n) = O(n)

Multiplicación de matrices

Se quiere calcular el producto C de dos matrices A,B cuadradas de dimensión n. El algoritmo directo tiene tiempo de ejecución en $\Theta(n^3)$, ya que cada uno de los n^2 elementos de C lleva tiempo $\Theta(n)$ en computarse.
La idea del algoritmo de Strassen es dividir las matrices en cuatro partes iguales, y resolver el producto original en base a operaciones sobre estas partes usando sumas y restas entre los componentes de $\Theta(n^2)$, en forma análoga al problema de multiplicar enteros grandes.
Se realizan solo 7 productos.



cada una de A11,...,C22 tiene dimensión n/2 × n/2 . Sumas y restas de estas matrices se puede computar en tiempo $\Theta(n^2)$

Se definen 7 matrices auxiliares donde cada una utiliza solo 1 multiplicación
Se obtiene el producto C sumando las distintas matrices auxiliares

```
def strassen(A,B)
    if(n < umbral)
        return A*B
    else
        //dividir
        calcular a11..a22 y b11..b22
        //conquistar
        m1 = strassen(...)
        ...
        m7 = strassen(...)
        //combinar
        calcular c11..c22  //sumas y restas O(n^2)
```

$$T(n) = \begin{cases} \Theta(n^3) & \text{si } n \text{ es pequeño} \\ 7T(n/2) + \Theta(n^2) & \text{sino} \end{cases}$$

$O(n^2)$ es por las sumas y restas.

$T(n) = \Theta(n^{\log 7})$ teorema maestro caso 1

Para las matrices cuyos n no son potencia de 2, es necesario completarlas con 0's hasta llegar a una potencia de 2

$$S(n) = \begin{cases} \Theta(1) & si \quad n \leq umbral \\ n + S(\frac{n}{2}) & si \quad n > umbral \end{cases}$$

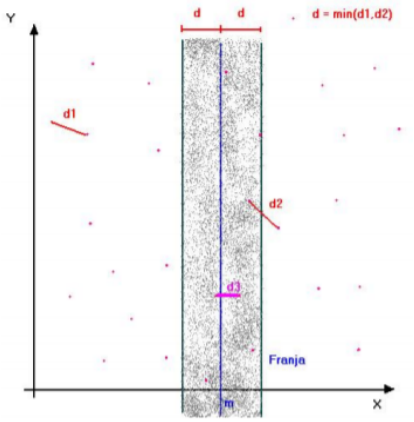
Creo 8 matrices de n/2 x n/2 que ocupan espacio n. Luego el árbol de recursión es de altura log n.
Por teorema maestro caso 3 S(n) = O(f(n)) = O(n)
S(n) = O(n).

Hay que completar con 0's las matrices, usa mucho espacio, si la matriz no es muy grande no vale la pena por la cantidad de constantes ocultas y el espacio ocupado.

Puntos más cercanos

Dado un conjunto de **n** puntos en el plano, se quiere encontrar el par de puntos más cercanos. El algoritmo básico compara todos los puntos entre sí y es de $O(n^2)$.
Se divide el conjunto de puntos en partes iguales, por medio de una recta **m**, según el eje x. Se encuentran las distancias mínimas en cada parte, d1 y d2. La menor distancia puede estar entre los dos conjuntos, entonces se crea una franja de ancho 2d alrededor de la recta m, siendo d = min(d1,d2), y se busca si existe una distancia menor que d en esa zona. Si hay una menor, se toma esa, si no, es d.

```
function MasCercanos(P[1..n],X,Y)//X = puntos ordenados por coord x
IF n es pequeño                  //Y = puntos ordenados por coord y
    RETURN algoritmoBasico(P)
ELSE
    //dividir
    m ::= punto medio de coord. x
    crear P1 y P2
    crear X1 y X2; crear Y1 y Y2
    //conquistar
    d1 ::= MásCercanos(P1,X1,Y1)
    d2 ::= MásCercanos(P2,X2,Y2)
    //combinar
    d ::= min(d1, d2)
    crear Franja con ancho 2d de m
    d3 ::= recorrido(Franja)
    RETURN min(d,d3)
```



para crear la franja creo arreglo Y' con los puntos de Y que están a distancia < d de la línea m ordenados por coord y. //**O(nlogn)**
para cada punto p, calculo su distancia con todos los puntos que están a distancia < d, que son 7.

$$T(n) = \begin{cases} \Theta(1) & \text{para } n \text{ pequeño} \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n \log n) & \text{sino} \end{cases}$$

T(n) = Θ(nlog² n) cambio de variables y crecimiento suave

Se ordena el arreglo antes de las llamadas recursivas según coordenadas para la creación eficiente de subinstancias. Agrega **O(n logn)** por única vez **T(n') = T(n) + O(nlogn).**
Se puede mejorar si se ordena P no solo por x, sino también por y, usando arreglos X e Y, de esta forma se elimina el ordenamiento en cada llamada recursiva, realizándose solamente una única vez al comienzo del algoritmo, y la creación de Franja se puede hacer entonces en tiempo lineal. Y el algoritmo resulta en **nlogn**.
T(n) = 2 T(n/2) + O(n) ⇒ T(n) = O(n logn)
S(n) = T(n/2) + O(n) ⇒ S(n) = O(n) teorema caso 3

Exponenciación modular

Dados enteros grandes m,n,z se quiere calcular **(m^n) mod z**. Esto es usado en criptografía para calcular claves y descifrar mensajes.
Usa las siguientes propiedades:

$x^y \bmod z = (x \bmod z)^y \bmod z$ y $xy \bmod z = [(x \bmod z)(y \bmod z)] \bmod z$

Algoritmo DYC:

Si **y** es par, **y = 2y'**, luego:

$$x^y \bmod z = x^{2y'} \bmod z = (x^2 \bmod z)^{y'} \bmod z$$

Si **y** es impar, **y = 2y' + 1**, luego:

$$x^y \bmod z = x^{2y'+1} \bmod z = x^{2y'} x \bmod z = [(x^{2y'} \bmod z)(x \bmod z)] \bmod z$$

Como la instancia se resuelve en base a la solución de un sólo subproblema, se trata de un caso de **simplificación**

```
function Expomod(x,y,z)
    a ::= x mod z
    IF y=1
        RETURN a
    ELSEIF (y es par)
        aux ::= a^2 mod z
        RETURN Expomod(aux,y/2,z)
    ELSEIF (y es impar)
        RETURN (a * Expomod(a,y-1,z)) mod z
```

La recurrencia de T(y) es :

$$T_{EXP}(y) = \begin{cases} \Theta(|x| + |z|) & \text{si } y = 1 \\ T_{EXP}(y/2) + \Theta(|z|^2) & \text{si } y \text{ es par} \\ T_{EXP}(y-1) + \Theta(|z|^2) & \text{si } y \text{ es impar} \end{cases}$$

Ni siquiera es eventualmente no decreciente. Para solucionarlo, se expande 1 vez el caso impar:

$$
\begin{aligned}
T_{EXP}(y) &= T_{EXP}(y-1) + \Theta(|z|^2) = \\
&= T_{EXP}(\lfloor y/2 \rfloor) + \Theta(|z|^2) + \Theta(|z|^2) = \\
&= T_{EXP}(\lfloor y/2 \rfloor) + \Theta(|z|^2)
\end{aligned}
$$

agrego O(z) de calcular el módulo.

T(y) ahora nos queda $T_{EXP}(y) \leq \begin{cases} \Theta(|x| + |z|) & \text{si } y = 1 \\ T_{EXP}(\lfloor y/2 \rfloor) + \Theta(|z|^2) & \text{si } y > 1 \end{cases}$

Esta nueva recurrencia, que sí es eventualmente no decreciente, tiene como resultado **T(y) = Θ(log y)**, teo maestro caso 2, considerando constante el costo de las multiplicaciones. **y** es el valor, no la longitud.

O(z2) es el orden de calcular el módulo, sumas y restas del orden de z. z es el tamaño máximo del número en el cual hago la aritmética. módulo y multi son del orden de z^2, suma y resta son de O(z). O(x + z) podría ser O(z^2).
La recurrencia está definida en base a y, poner cualquier cosa que no dependa de y, es lo mismo que poner O(1) en el caso base.

S(y) = O(log y) teo caso 2.

Transformada de Fourier

Toma una función representando un patrón basado en tiempo y la descompone en función de varios ciclos. Se usa en video y audio digital, en imágenes, sonares, etc.
La señal original se representa como un polinomio de señales más sencillas.

Formas de representar un polinomio:

1. **Por coeficientes:** sea A(x) un polinomio, se puede representar por coeficientes como A(x) = a0 +a1x +a2x 2 +...+an−1x n−1 de grado n−1, siendo los coeficientes ai ∈ F y F un campo cualquiera (por ejemplo Q,R,C,...)
2. **Por raíces:** A(x) se puede representar también por sus raíces ri ,1 ≤ i ≤ n−1, entonces A(x) = c(x −r1)...(x −rn−1)
3. **Por muestras:** A(x) grado-acotado por n tomando n muestras A(x) = {((x0,y0),(x1,y1),...,(xn−1,yn−1)}

Operaciones:

1. **evaluación:** dados A(x) y x0, encontrar y0 = A(x0)
2. **suma:** dados A(x) y B(x), encontrar C(x) = A(x) +B(x)
3. **multiplicación:** dados A(x) y B(x), encontrar C(x) = A(x)×B(x)
4. **convolución:** dados A(x) y B(x), encontrar C(x) = A(x)⊗B(x)

Implementaciones de polinomios:

	COEFICIENTES	RAÍCES	MUESTRAS
EVALUACIÓN	O(n)	O(n)	$O(n^2)$
SUMA	O(n)	-	O(n)
PRODUCTO	$O(n^2)$	O(n)	O(n)

FFT permitirá la **transformación Coeficientes↔Muestras** en tiempo **O(n.logn)**, y acotar así también los tiempos de las operaciones no importa su representación.

El algoritmo normal de transformación tiene $O(n^2)$.
La clave es elegir buenos puntos de muestras. Sería necesario que el tamaño de los puntos de muestras X disminuya al mismo tiempo que los coeficientes A, para eso se usa el conjunto de raíces n-ésimas de la unidad. Esto reduce los tiempos de evaluación del polinomio.

Esto permite que $T_{FFT}(n) = 2T_{FFT}(n/2) + \Theta(n) \in \Theta(n \log n)$

Programación Dinámica

Resuelve problemas a través de combinar soluciones a subproblemas. Comienza resolviendo las instancias más simples de los problemas, y guardando sus resultados en alguna estructura de datos especial. Para construir soluciones de instancias más complejas, se divide la instancia en subproblemas más simples y se recuperan los resultados ya calculados de la estructura de datos. PD se aplica cuando los subproblemas no son independientes entre sí, es decir los subproblemas tienen subsubproblemas en común. Esto se denomina superposición de subproblemas.
DyC resuelve las instancias siempre dividiendo, sin importar cálculos previos. En este contexto, se resolverían varias veces el mismo subproblema (como el primer algoritmo para Fibonacci). DYC se usa cuando no hay superposición de subproblemas, o es casi nula. Un algoritmo de PD resuelve cada subproblema una vez, guardando sus resultados y evitando el trabajo de calcularlo otra vez.
DYC es una técnica top-down; PD por el contrario es bottom-up.

Principio de optimalidad: la estructura de una solución optimal a un problema debe contener soluciones optimales a los subproblemas

Superposición de subproblemas: el “espacio” de subproblemas debe ser pequeño en el sentido de que los subproblemas se repiten una y otra vez, en vez de generar nuevos subproblemas

Estos problemas de PD están dados en un contexto simplificado, en la práctica se pueden dar en situaciones más diversas, a veces tiene sentido encontrar solo el valor óptimo y a veces también se necesita devolver la secuencia.

(Coeficientes binomiales y prob de ganar serie)

Coeficientes binomiales

no se trata de un problema de optimización, pero la solución está formada por combinación de soluciones de subproblemas además, claramente se ve superposición de subinstancias:
C(5,3) = C(4,3) +C(4,2) = (C(3,3) +C(3,2)) + (C(3,2)+C(3,1)) = …

se puede usar una tabla para guardar resultados intermedios, donde la entrada (i,j) guarda el número C(i,j)

```
function CoeficientesBinomiales(n,k) //calcula por filas
    array C[1..n,1..n]
    para todo k C[k,0] ::= 1; C[k,k] ::= 1;
    FOR i ::= 1 TO TO n
        FOR j ::= 1 TO min(i,k)
            C[i,j] ::= C[i-1,j-1]+C[i-1,j]
    RETURN C[n,k]
```

Su tiempo y espacio es claramente de **Θ(nk)**. se puede modificar el algoritmo para que sólo use espacio Θ(k).
No hace falta almacenar toda la tabla, se puede mantener un vector de longitud k, que represente la fila actual, y actualizar este vector de izquierda a derecha. Nos queda espacio **Θ(k).**

Probabilidad de ganar serie

Dos equipos A y B deben jugar hasta 2n−1 juegos, siendo el ganador el primer equipo que llega a n victorias. Para cada juego existe una probabilidad p de que gane el equipo A, y una probabilidad q = 1−p de que gane el equipo B. Esta probabilidad es fija para todos los juegos, e independiente de los resultados anteriores. Se quiere encontrar la probabilidad de que el equipo A gane la serie.
Se define P(i,j) como la probabilidad de que A gane la serie dado que le faltan i victorias, mientras que a B le faltan j victorias.
Entonces el valor buscado es P(n,n)

Se genera la recurrencia

$$P(i,j) = \begin{cases} 1 & \text{si } i = 0 \text{ y } j > 0 \\ 0 & \text{si } j = 0 \text{ y } i > 0 \\ pP(i-1,j) + qP(i,j-1) & \text{si } i > 0 \text{ y } j > 0 \end{cases}$$

sea k = j +i. El algoritmo de cálculo recursivo de P tomaría tiempo:

$$\begin{aligned} T(1) &= c \\ T(k) &\leq 2T(k-1) + d \end{aligned}$$

la solución (usando la ecuación característica) es de O(2 k), lo que equivale a O(4 n) si i = j = n.
Esta estructura del problema es similar a la de los coeficientes binomiles tomando P(i,j) como C(i +j,j)

Es posible mejorar este tiempo en forma similar al triángulo de Pascal, calculando P por filas, columnas o diagonales

Su tiempo y espacio es de $\Theta(n^2)$ se puede hacer la misma modificación que en el caso anterior para que use espacio en Θ(n)

Problema del cambio

se tiene que dar N centavos de cambio, usando la menor cantidad entre monedas de denominaciones d1,d2,d3,...,dn. Se supone cantidad ilimitada de monedas de cada denominación.

Se define C[i,j] como la menor cantidad de monedas entre d1,d2,...,di para pagar j centavos
la solución está entonces C[n,N].

Se satisface el principio de optimalidad. Si la solución optimal C[n,N] incluye una moneda de d_n entonces deberá estar formada por la solución optimal $C[n,N-d_n]$. En cambio si no incluye ninguna moneda de dn, su valor será la solución optimal a C[n−1,N] .

```
function Cambio(D[1..n],N)
  array C[1..n,0..N] ::= 0
    FOR i::=1 TO n
      FOR j::=1 TO N
        CASE
          i=1 y j<d[i]: C[i,j]::= maxint
          i=1 y j>=d[i]: C[i,j]::= 1+C[i,j-d[i]]
          i>1 y j<d[i]: C[i,j]::= C[i-1,j]
          i>1 y j>=d[i]: C[i,j]::= min(C[i-1,j], 1+C[i,j-d[i]])
  RETURN C[n,N]
```

$$C[i,j] = \begin{cases} 0 & \text{si } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } 0 < j < d_i \\ 1 + C[i, j - d_i] & \text{si } i = 1 \text{ y } j \geq d_i \\ C[i-1, j] & \text{si } i > 1 \text{ y } j < d_i \\ \min(C[i-1, j], 1 + C[i, j - d_i]) & \text{si } i > 1 \text{ y } j \geq d_i \end{cases}$$

El tiempo y el espacio es de **Θ(nN)**.
Este algoritmo sólo encuentra el mínimo número de monedas necesarias, pero no dice cuáles son. Para encontrar las monedas que forman el cambio, se analiza cada la entrada C[i,j]: si es igual a C[i−1,j] entonces no se usan monedas d_i; en caso contrario se usa una moneda d_i más las monedas de C[i,j−d_i] Partiendo de C[n,N] y retrocediendo por fila, o por columna, de acuerdo a su valor, hasta llegar a C[0,0], se obtienen las C[n,N] monedas que forman el cambio. Este recorrido agrega tareas por tiempo Θ(n+C[n,N]) al algoritmo original.

Cuando tengo el cambio voy una fila delante o a la misma fila unas posiciones hacia la izquierda, entonces en esos caso puedo optimizar y el espacio me queda del orden del monto a dar el cambio, pero si tengo que guardar además las decisiones que tome, tengo que tener la matriz. Si optimizo el espacio, no puedo dar las monedas que dan la solución.

La dependencia del tiempo y el espacio de ejecución en un dato de entrada **N no es buena** porque puede ser arbitrariamente **grande.**

Usos: Cajeros, una fábrica para crear un producto de varias formas y ver cual me da el menor costo posible.

Problema de los cortes

Varilla de **n** centímetros de longitud, y los precios pi correspondientes a varillas de i = 1,2,...,n centímetros. Averiguar cómo cortar la varilla de forma de maximizar la ganancia obtenida G por la venta de todas las partes.
Sea k la cantidad de cortes, para k ≥ 1, G(n) = p_{i1} +G(n−i_1) lo que da al problema una estructura recursiva.
Optimalidad: se demuestra por inducción generalizada que la solución óptima G*(n)=pi +G*(n-i) para algún **i ≤ n**. Paso inductivo demostrado por absurdo, se supone que G*(n)=pi+G*(n-i) no es optimal, y existe un G'(n)=pi+G'(n-i) mejor. Cuando se llega a este paso no queda otra que G'(n-i) sea igual a G*(n-i), por la HI.

```
function Cortes(p[1..m], n) - con m>=n
    array optimal[0..n] ::= 0
    FOR j::=1 TO n
        aux ::= -maxint
        FOR i::=1 TO j
            aux ::= max(aux,p[i]+optimal[j-i])
        optimal[j] ::= aux
    RETURN optimal[n]
```

T(n) = O(n^2) por los dos ciclos anidados.
Espacio = O(n) por el arreglo auxiliar.
El algoritmo solo devuelve el valor optimal, para saber cuáles cortes dan ese valor es necesario usar otro arreglo que contenga para cada posición i cual es la posición j < i en la que hay que cortar, sin cambio en el tiempo ni en el espacio

Usos: Maximizar el valor de las ventas de un material, reducir los desperdicios.

Problema de la mochila

N objetos indivisibles con peso w_i y valor v_i. Mochila con peso máx W.
Encontrar la carga de la mochila que maximice el valor sin pasar la capacidad.

Optimalidad: The parallel with the problem of making change is close. As there, the principle of optimality applies. We may fill in the table either row by row or column by column. In the general situation, V[i,j] is the larger (since we are trying to maximize value) of V[i-1, j] and V[i -1, j-w_i] + v_i. The first of these choices corresponds to not adding object i to the load. The second corresponds to choosing object i, which has for effect to increase the value of the load by v_i and to reduce the capacity available by w_i

V[i,j] = el máximo valor de una carga de peso máximo **j** con los objetos numerados 1,2,...,i.
V[i,j] es el max(V[i-1,j] , (V[i-1, j-w_i] + v_i)) La primer opción corresponde a no agregar el objeto i, y la segunda corresponde a agregarlo, aumentando el valor en v_i y reduciendo la capacidad en w_i.

$$V[i,j] = \begin{cases} 0 & \text{si } j = 0 \\ -\infty & \text{si } i > 0 \text{ y } j < 0 \\ \max(V[i-1,j], & \\ \ v_i + V[i-1,j-w_i]) & \text{si } i > 0 \text{ y } j > 0 \end{cases}$$

Caso j < 0: estoy en solución parcial donde la carga máxima es 2 y tengo un objeto que pesa más que 2, la resta me da negativa. nunca voy a llamar a la mochila con j<0, puede suceder en un caso intermedio

```
function mochila(v[1..n],w[1..n],W)
  array maxValor[1..n,0..W] ::= 0
    FOR i::=1 TO n
      FOR j::=1 TO W
          CASE
            j=0: V[i,j] = 0
            j<0: V[i,j] = -maxint
            j>0: V[i,j] = max(maxValor[i-1,j],v[i]+maxValor[i-1,j-w[i]])
  RETURN maxValor[n,W]
```

Este algoritmo es muy similar al del problema del cambio.
T(n) = Espacio = Θ(nW)
Para calcular cuáles objetos componen la carga optimal se puede hacer un recorrido adicional desde C[n,W] hasta C[0,0] de Θ(n+W)

Usos: se usa en logística, para cargar camiones, aviones, se utilizan otras variantes, volumen con peso con valor.

La dependencia del tiempo y el espacio de ejecución en un dato de entrada **N no es buena** porque puede ser arbitrariamente **grande.**

Camino más corto

Consiste en encontrar el camino con la menor distancia entre todo par de nodos en un grafo.
Se puede usar Dijkstra **n** veces cambiando el nodo de origen, pero no sirve para arcos con peso negativo.
Optimalidad: si **k** es un nodo en el menor camino entre **i** y **j**, entonces ese camino está formado por el menor camino de **i** a **k** y el menor camino de **k** a **j.** Entonces, para ir calculando cada D[i,j] se pueden considerar el conjunto de nodos intermedios 1,..., k que pueden ir formando parte de posibles caminos intermedios. Para cada **k**, existen dos alternativas: o **k** pertence al menor camino entre **i** y **j**, o no pertenece, y es el mismo que para 1,..., k −1.
Superposición de instancias: Sea D[i,j,k] la menor distancia entre i y j que tiene como nodos intermedios a 1,2,..., k. se debe comparar el camino más corto obtenido hasta entonces (con nodos intermedios 1,..., k −1), con el camino que va desde **i** hasta **k**, y luego de **k** a **j**, también sólo con nodos intermedios 1,..., k −1.

```
function Floyd(G[1..n,1..n])
    D::=G //inicialmente contiene los pesos de los arcos (i,j)
    P::=[n][n] = 0
    FOR k::=1 TO n
        FOR i::=1 TO n
            FOR j::=1 TO n
                IF (D[i,k]+D[k,j] < D[i,j])
                    D[i,j]::=D[i,k]+D[k,j]
                    camino[i,j]::= k   //opcional p camino
    RETURN D,camino
```

$$D[i,j,k] = \begin{cases} G[i,j] & \text{si } k = 0 \\ \min(D[i,j,k-1], & \\ \ D[i,k,k-1] + D[k,j,k-1]) & \text{sino} \end{cases}$$

k-1 = iteración anterior

T(n) = Θ(n^3) 3 bucles anidados
S(n) = Θ(1) trabajo sobre las matrices de salida.
Para saber el camino se debe tener una matriz adicional P y cada vez que se modifica D[i,j], se actualiza P[i,j] con k.

Usos: un grafo sin pesos puede ser usado para representar una relación entre los nodos. Entonces para determinar si existe un camino entre un dado par de nodos es necesario calcular la clausura transitiva de la relación. Para esto se asigna peso 1 para los arcos que existen, y se calculan mediante Floyd-Warshall los caminos mínimos del grafo. la clausura transitiva se usa en compiladores para poder saber cuáles son los terminales iniciales para todos los símbolos no-terminales de una gramática dada

Multiplicación de matrices en cadena

se tienen n matrices M1,M2,...,Mn, no necesariamente cuadradas, y se quiere encontrar la mejor manera de hallar su producto M1M2 ...Mn. Cada matriz M_i es de tamaño $d_{i-1}d_i$.
El orden en que se realizan los productos es muy importante, el problema entonces consiste en encontrar todas las parentizaciones posibles para M1,M2,...,Mn, evaluar la cantidad de productos necesarios, y obtener el menor entre todos ellos.

La función a optimizar es la cantidad de productos de reales necesarios para multiplicar una secuencia de matrices. Este valor depende de la cantidad de productos necesarios para multiplicar subsecuencias de matrices.

Se define m_{ij} ,i ≤ j como la mínima cantidad de productos necesarios para calcular Mi*...*Mj. Claramente, si **i = j** entonces $\mathbf{m_{ii} = 0}$ y si **j = i +1** entonces $\mathbf{m_{ii}+1 = d_{i-1}*d_i*d_{i+1}}$

En general, si i < j: $m_{ij} = \min_{i \le k < j}^{k} (m_{ik} + m_{(k+1)j} + d_{i-1} d_k d_j)$

Optimalidad: if the best way of multiplying all the matrices requires us to make the first cut between the i-th and the (i+1)-st matrices of the product, then both the subproducts $M_1M_2...M_i$ and $M_{i+1}M_{i+2}...M_n$ must also be calculated in an optimal way.

```
function MultMatrices(d[0..n])
    array m[1..n,1..n]::=0;
    array bestk[1..n,1..n]::=0;
        FOR s::=1 TO n-1
            FOR i::=1 TO n-s;
                menor::= +maxint
                FOR k::=i TO i+s-1
                    tmp::=m[i,k]+m[k+1,i+s]+d[i-1]*d[k]*d[i+s]
                    IF tmp<menor
                         menor::=tmp
                         bestk[i,j] = k
                m[i,i+s]::=menor
    RETURN m[1,n], bestk
```

Cuando computamos m_{ij}, guardamos en bestk[i,j] el valor de k que determinó su menor valor. Al final del algoritmo, bestk[1,n] nos dice dónde realizar el primer corte.

Tomamos como barometro cualquier sentencia del ciclo interno:

$$T(n) = \sum_{s=1}^{n-1}\sum_{i=1}^{n-s}\sum_{k=i}^{i+s-1} c = \sum_{s=1}^{n-1}\sum_{i=1}^{n-s} sc = c\sum_{s=1}^{n-1}\sum_{i=1}^{n-s} s = c\sum_{s=1}^{n-1}(n-s)s = nc\sum_{s=1}^{n-1} s - c\sum_{s=1}^{n-1} s^2 =$$

$$= n\frac{c}{2}(n-1)n - (n-1)n(2n-1)\frac{c}{6} = \frac{c}{6}n^3 - \frac{c}{6}n \quad \in \Theta(n^3)$$

S(n) = Θ(n^2).

Usos: triangulación de polinomios

Triangulación optimal de polígonos

Se tiene un polígono convexo (v_0,v_1,..., v_{n-1}) de n lados, siendo v_i los vértices y v_{i-1}-v_i el lado i. Se quiere encontrar una triangularización optimal, de acuerdo a algún criterio dado.
Una cuerda v_iv_j es el segmento formado por un par de vértice no adyacentes.
Toda cuerda divide al polígono en dos subpolígonos.
Una triangularización es un conjunto de cuerdas que dividen al polígono en triángulos disjuntos.
Si se tiene un peso w($v_iv_jv_k$) para cada triángulo $v_iv_jv_k$, entonces una triangularización optimal de un polígono es una triangularización que minimiza la sumatoria de los pesos de los triángulos resultantes.
El peso suele ser el perimetro w($v_iv_jv_k$) = |v_iv_j|+|v_jv_k|+|v_kv_i|.

Este problema se puede reducir a la multiplicación de matrices. Se define una reducción TRIANGULARIZACIÓN → CADENAMATRICES dado un polígono (v_0,v_1,..., v_{n-1}), se establece una correspondencia entre los lados (excepto v_{n-1}-v_0) y "matrices" A_i, cuyo "tamaño" es $v_{i-1}v_i$ y con "tiempo de multiplicación" $w(v_iv_jv_k)$.
Luego cada forma de multiplicar las matrices A_1A_2 ...A_{n-1} corresponde a una triangularización del polígono.

```
function triangulacion(w[0..n])
    array m[1..n,1..n]::=0;
        FOR s::=1 TO n-1
            FOR i::=1 TO n-s;
                menor::= +maxint
                FOR k::=i TO i+s-1
                    tmp::=m[i,k]+m[k+1,i+s]+w(v[i],v[j],v[k])
                    IF tmp<menor
                         menor::=tmp
                m[i,i+s]::=menor
    RETURN m[1,n]
```

T(n) = Θ(n^3)

Subsecuencia común más larga

Dadas una secuencia X = (x1,x2,..., xm), otra secuencia Z = (z1,z2,..., zk) es una subsecuencia si existe una secuencia creciente de índices i1,i2,...,ik tal que $x_{ij} = z_j$ para todo j,1 ≤ j ≤ k.
dadas dos secuencias X,Y el problema de la subsecuencia común más larga (LCS) es el problema de encontrar una subsecuencia común de longitud máxima para X,Y

Optimalidad:

Teorema 1

Sean $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Luego si $Z = \langle z_1, z_2, \ldots, z_k \rangle$ es LCS de X, Y y

- $x_m = y_n$, *entonces* Z_{k-1} *es LCS de* X_{m-1}, Y_{n-1}
- $x_m \neq y_m$ *y* $z_k \neq x_m$, *entonces Z es LCS de* X_{m-1}, Y
- $x_m \neq y_m$ *y* $z_k \neq y_n$, *entonces Z es LCS de* X, Y_{n-1}

Se prueban los 3 puntos por contradicción, llegando a que Z no es LCS de X,Y.

C[i,j] es la longitud de una LCS de X_i e Y_j.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Se ve claramente la superposición de subinstancias.

```
function LCS(X[1..m], Y[1..n])
```

```
array C[1..m,1..n]::=0
FOR i::=1 TO m
    FOR j::=1 TO n
        IF X[i]==Y[j]
            C[i,j]::=C[i-1,j-1]+1
        ELSIF C[i-1,j]>=C[i,j-1]
            C[i,j]::=C[i-1,j]
        ELSE
            C[i,j]::=C[i,j-1]
RETURN C[m,n]
```

T(n) = O(mn) por el doble bucle.
S(n) = O(mn)

Si quiero recuperar la LCS, tengo que devolver la matriz entera. La función de reconstrucción reconstruye las decisiones tomadas cuando calculaba la tabla `c`. Si el último carácter en el prefijo son iguales, entonces deberían ser LCS. Si no, debería verificar cuál tiene el mayor LCS almacenando X_i y Y_i, y tomando la misma decisión. Solamente elige uno cuando son igual de largos. Se llama la función con `i=m` y `j=n`.

```
function backtrack(C[0..m,0..n], X[1..m], Y[1..n], i, j)
    if i = 0 or j = 0
        return ""
    else if X[i] = Y[j]
        return backtrack(C, X, Y, i-1, j-1) + X[i]
    else
        if C[i,j-1] > C[i-1,j]
            return backtrack(C, X, Y, i, j-1)
        else
            return backtrack(C, X, Y, i-1, j)
```

Usos: en bioinformática, es frecuente la necesidad de comparar el ADN de dos o más organismos.
En git para ver los cambios.

Grafos

Etiquetado de nodos

para recorrer grafos, se etiquetarán dinámicamente los nodos como:

- nodos blancos: todavía no han sido visitados
- nodos grises: ya han sido visitados, pero no se ha controlado la visita a todos sus adyacentes
- nodos negros: ya han sido visitados, al igual que todos sus adyacentes

Esta caracterización implica que ningún nodo negro tiene un nodo blanco como adyacente.

Foresta: subgrafo de G formado por todos los arcos utilizados en el recorrido. se representa a través de un arreglo adicional **padre[1..n]** donde cada nodo guarda su antecesor en la foresta; si padre[i] = 0 entonces el nodo es una raíz.

Los recorridos permiten clasificar los arcos del grafo en las siguientes categorías:
- **arcos de la foresta** son los arcos (padre[v],v) utilizados en el recorrido
- **arcos hacia atrás (B)** son arcos (u,v) en donde **v** es un ancestro de **u** en la foresta del recorrido
- **arcos hacia adelante (F)** son arcos (u,v) en donde **v** es descendiente de **u** en la foresta del recorrido
- **arcos cruzados (C)** son los demás arcos que no entran en las otras categorías. Los extremos pueden estar en el mismo árbol o en árboles diferentes

Si el grafo es no dirigido, como cada arco es considerado dos veces esta clasificación puede ser ambigua. Se toma la primera de las categorías posibles según el orden dado.

BFS

```
PROCEDURE bfs(G=N,A)
    FOR cada vértice v en N
        color[v]::=blanco
    Q.ColaVacía()
    FOR cada vértice v en N
        IF color[v]=blanco
            color[v]::=gris
            Q.insertar(v)
            visitarBF(G,Q)

PROCEDURE visitarBF(G,Q)
    WHILE no Q.esVacía()
        u::=Q.primero()
        IF existe (u,w) tq color[w]=blanco
            color[w]::=gris
            padre[w]::=u; nivel[w]::=nivel[u]+1
            Q.insertar(w)
        ELSE
            color[u]::=negro
            Q.sacarDeCola()
```

$$T_{BF}(n) \leq \sum_{i=1}^{n}(c + T_{visitarBF}(n)) = \sum_{i=1}^{n}(a + O(a \times n)) \in O(n^4)$$

Es una cota muy mala.
Se puede probar por inducción sobre el número de iteraciones (1) que todo nodo es coloreado blanco, gris y negro exactamente una vez, y en ese orden. (2) que todos los nodos en Q son grises y solamente están una vez en Q. (3) que los controles de adyacencia se hacen exactamente una vez por cada arco. Esto resulta $\mathbf{T_{BF}(n) = \Theta(max(n,a))}$ tomando como barómetros las operaciones sobre la E.D. Cola

El **Nivel** de cada nodo es la distancia (en cantidad de arcos) que deben recorrer en la foresta para llegar desde la raíz al nodo correspondiente se representa por un arreglo adicional nivel[1..n], inicializado en 0. Al finalizar un recorrido BFS, nivel[v] contiene la mínima distancia (en cantidad de arcos) desde la raíz del árbol de v en la foresta de recorrido hasta v
Lema 4 Si el grafo es **no dirigido**, entonces un recorrido BF sólo genera arcos de la foresta y cruzados.
Lema 5 Si el grafo es **dirigido**, entonces un recorrido BF no genera arcos hacia adelante.

DFS

```
PROCEDURE dfs(G=N,A)
    FOR cada vértice v en N
        color[v]::=blanco
    P.pilaVacía(); tiempo::=0
    FOR cada vértice v en N
        IF color[v]=blanco
            color[v]::=gris; tiempo++; d[v]::=tiempo
            P.apilar(v)
            visitarDF(G,P)

PROCEDURE visitarDF(G,P)
    WHILE no P.esVacía()
        u::=P.tope()
        IF existe (u,w) tq color[w]=blanco
            color[w]::=gris
            tiempo++; d[w]::=tiempo
            P.apilar(w)
        ELSE
            color[u]::=negro
            tiempo++; f[u]::=tiempo
            P.desapilar()
```

Todo nodo es coloreado blanco, gris y negro exactamente una vez, y en ese orden. Todos los nodos en P son grises y solamente están una vez en P. Los controles de adyacencia se hacen exactamente una vez por cada arco. **T(n) = Θ(n+a).**

Se puede obtener la foresta pero la numeración nivel[u] no tiene sentido.
Sí es posible en este tipo de recorrido numerar a los nodos de acuerdo al tiempo del evento de **descubrimiento** (numeración preorden - cuando se marca gris - arreglo d[1..n]), o de **finalización** (numeración postorden - cuando se marca negro - arreglo f[1..n]). El descubrimiento coincide con la inserción del nodo en la pila; la finalización con su eliminación.

Lema 6: Sea G = (N,A) un grafo, y d[],f[] la numeración de descubrimiento y finalización obtenida mediante un recorrido DF. Entonces

- (u,v) es de la **foresta** o **F** si y sólo si d[u] < d[v] < f[v] < f[u]
- (u,v) es **B**(hacia atrás) si y sólo si d[v] < d[u] < f[u] < f[v]
- (u,v) es **C**(cruzado) si y sólo si d[v] < f[v] < d[u] < f[u]

Teorema 7: En un recorrido DF de un grafo no dirigido G, todos sus arcos son de la foresta o hacia atrás.
Demostración. Sea (u,v) un arco del grafo, y supongamos que u es descubierto antes que v. Luego v es descubierto y finalizado antes de finalizar u. Si el nodo v tiene como padre a u entonces (u,v) es un arco de la foresta; si el nodo v tiene otro padre, entonces (u,v) es un arco hacia atrás. En forma similar se prueba si v es descubierto antes que u.

Lema 11: Un grafo dirigido G es **acíclico** si y sólo si cualquier recorrido DF de G no produce arcos hacia atrás

	calcula	arcos en g. no dir.	arcos en g. dir.	propiedades
BFS	`padre[]` `nivel[]`	de la foresta C	de la foresta B C	conectitud (no dir.) distancia mínima

DFS	`padre[]`
	`d[]`
	`f[]`
		de la foresta
		B
			de la foresta
			B
			F
			C
				conectitud (no dir.)
				caminos blancos
				caracteriz. dags

Ordenamiento topológico

Dado un grafo dirigido acíclico G, un orden topológico es un ordenamiento lineal de sus nodos de forma que si el arco (u,v) ∈ G entonces u aparece antes de v en el ordenamiento. Si el grafo tiene ciclos, entonces tal ordenamiento no existe.
El orden topológico es usado para planificar una serie de acciones que tienen precedencias: cada nodo representa una acción, y cada arco (u,v) significa que la acción u debe ejecutarse necesariamente antes de v.

```
PROCEDURE OrdenTopológico(G)
    lista L
    DFS(G,L) computa los finishing times de cada vértice v y a medida que
       cada vértice finaliza, se inserta al inicio de la lista L
    RETURN L
```

T(n) = Θ(n+a). Ya que es el T(n) del DFS y toma O(1) insertar al inicio de una lista.
S(n) = S(dfs) = O(n)

Componentes fuertemente conexas

dado un grafo dirigido G = (N,A), un componente fuertemente conexo (CFC) es un conjunto U ⊆ N maximal tal que para todo u,v ∈ U existe en G un camino de u a v y viceversa.
Problema: encontrar todos los CFC de un grafo dirigido G.

Sea G un grafo, su grafo traspuesto G^T se define como $G^T = (N,A^T)$ donde A^T={ (u,v) | (v,u) ∈ A }
T(crear C^T) = O(N + A) en lista de ady o $O(N^2)$ en matriz.

El algoritmo para CFC hace dos recorridos: uno del grafo G y otro del grafo G^T en el segundo recorrido, los nodos se consideran en orden decreciente de f[].

```
PROCEDURE CFC(G)
    array f[1..n] //finalización de los nodos
    DFS(G,f)
    calcular GT // el traspuesto de G
    array padre[1..n]
    DFS(GT, padre, f) //padre es la foresta del recorrido, en el loop
                      //principal del DFS, considero los vértices en orden
                      //decreciente de f[v]
    RETURN padre
```

Cada árbol de la foresta representado en padre contiene exactamente los elementos de un CFC
T(n) = O(n+a) de calcular el G^T con lista de ady, $O(n^2)$ con matriz
S(n) = O(GT) = O(n^2) con matriz o O(n+a) con listas.

Usos: Para resolver el 2SAT. Una instancia de 2SAT es insatisfacible si y sólo si existe una variable v tal que v y su complemento están contenidos en el mismo CFC del gráfico de implicancias de la instancia.

Caminos más cortos

Sea G = (N,A) un grafo dirigido, con pesos no negativos, en donde existe un nodo distinguido llamado origen
Camino más corto desde único origen, hacia único destino, para todo par de nodos, se resuelven con el de origen único.

consideraremos el grafo representado con una matriz de adyacencia G, con nodos 1..n, y al nodo origen etiquetado con 1. G[i,j] contiene el peso del arco (i,j) si (i,j) ∈ A, o G[i,j] = ∞ si el arco no existe
atacaremos primero el problema de encontrar las distancias mínimas, y después el de los caminos que la implementan datos de entrada: G[1..n, 1..n] el grafo; se supone que el origen es el nodo 1

Dijkstra greedy
consiste en mantener en S al conjunto de nodos cuyas distancias mínimas ya se conoce, y en C a aquellos que todavía falta calcular en cada paso se selecciona el elemento u de C más cercano al origen, y se relajan los arcos que salen de u

```
array dist[n]
array p[n] = 1
FOR i ::= 1 TO n
    dist[i] ::= G[1,i]
S ::= {1}; C ::= {2, ..., n}
FOR j ::= 1 TO n-2
    u ::= el elemento de C que minimiza dist[u]
    S ::= S + {u}; C ::= C-{u}
    FOR cada (u,v) en A
        IF dist[v]>dist[u]+G[u,v]
            dist[v] ::= dist[u]+G[u,v]
            p[v] ::= u
RETURN dist,p
```

para obtener los caminos más cortos se necesita un arreglo p[1...n], inicializado en 1, donde p[i] contiene el último nodo en el camino más corto entre 1 y i.

Usamos como parámetro alguna sentencia dentro del for interno, cuya ejecución está acotada en total por la cantidad de arcos.

$$T_{\text{Dijsktra}}(n) = an + b + \sum_{j=1}^{n-2}\sum_{k=1}^{n-1} c + \sum_{j=1}^{n-2} d + \sum_{j=1}^{a} e = \in \Theta(n^2)$$ con a <= n^2

S(n) = O(n)

Si el grafo es ralo (a << n^2) conviene usar una lista de adyacencias.
para seleccionar el próximo candidato se puede usar un heap; pero será necesario actualizarlo cada vez que se elimina el elemento y cada vez que se modifica alguna distancia en d. el tiempo total es de Θ((a+n)log n) = Θ(alog n). ¿por qué?

Árboles minimales de cubrimiento

un subgrafo de cubrimiento es un subgrafo G' = (N,A') con A' ⊆ A, que también es conexo. Es minimal si contiene la menor cantidad de arcos entre todos los cubrimientos de G.

Sea G un grafo no dirigido, conexo y con pesos, entonces todo subgrafo minimal de cubrimiento de G es un árbol, de n-1 arcos, y siempre posee al menos un árbol minimal de cubrimiento.

Un algoritmo greedy para solucionar este problema tendría como conjunto de candidatos a A. Un conjunto es una solución si contiene n−1 arcos (es un árbol y cubre todos los nodos). El control de viable se puede realizar controlando por la no existencia de ciclos (los candidatos deben siempre formar un árbol). De acuerdo a distintas funciones de selección se definen dos algoritmos para este problema: Kruskal y Prim.

Kruskal

Algoritmo greedy que encuentra un árbol mínimo de cubrimiento.
Selecciona en cada iteración el menor de los arcos todavía no considerados. Si es viable esta selección, se incluye en la solución parcial. En caso contrario, es descartado.
la función de selección elige arcos sin considerar la conexión con los arcos precedentes

```
def kruskal(g):
    heap = arcos ordenados en orden creciente por peso // O(a loga)
    disjointSet = DisjointSet(g.N) //O(n)
    arcos = []
    foreach arco (u,v) en heap: //O(a)
        nodo1 = find-set(u)
        nodo2 = find-set(v)
        if(nodo1 != nodo2):
            union(nodo1, nodo2)
            arcos = arcos + {(u,v)}
    return arcos
```

```
def kruskal(g):
    ordenar A en L; n::= |N|; T::={}      // Θ(alog a)
    D.initiate(N)
    REPEAT //ciclo greedy                 //Θ(n)
        (u,v)::= primero(L) // lo remueve
        compu::= D.find(u)
        compv::= D.find(v)
        IF (compu != compv)
            D.merge(u,v)
            T::= T + {(u,v)}
    UNTIL (|T|=n-1)
    RETURN T
```

si G es conexo n−1 ≤ a ≤ n(n−1)/2. K llamadas a operaciones find() y merge() en una E.D. de conjuntos disjuntos de n elementos lleva tiempo de O(K log∗ n) log∗ n ∈ O(log n).
T(G) = Θ(a logn)
S(n) = Θ(a)
Si uso un heap(implementación de arriba) y ordeno en cada iteración, se reducen las constantes ocultas.

Usando compresión de caminos y unión by rank se reducen los tiempos.
T(n) = Θ(a log a) + Θ(n) + O(a log* n) El bucle ejecuta **a** operaciones de find y unión que toma O(a α(n)) que es una función de crecimiento muy suave.

Prim

El algoritmo de Prim se caracteriza por hacer la selección en forma local, partiendo de un nodo seleccionado y construyendo el árbol en forma ordenada.

```
Prim(G)
    T ::= {}
    B ::= {un nodo de N}
    WHILE (B != N) // Ciclo greedy
        encontrar (u,v) tal que peso((u,v)) sea mínimo con u en B y v en
N-B
        T ::= T+{(u,v)}
        B ::= B+{v}
    RETURN T

Prim(G)
    B ::= {1}; T ::= {}
    FOR i ::=2 TO n
        distB[i] ::= G[1,i]
        IF (G[1,i]<MaxInt) THEN
            masCercano[i] ::= 1
        ELSE
            masCercano[i] ::= 0
    REPEAT // Ciclo greedy
        k::= nodo con distB[k]>0 y que minimice distB[k]   //O(n)
        T::= T+{(k,masCercano[k])}
        B::= B+{k}; distB[k]::= 0
        FOR cada j adyacente a k en G                      //O(2a)
            IF G[k,j]<distB[j] //RELAXATION
                dist[j]::= G[k,j]
                masCercano[j] ::= k
    UNTIL |B|=n                                            //O(n)
```

$T(n) = \Theta(n^2)$ ya que $n-1 <= a <= n^2$
$S(n) = \Theta(n)$
$T(kruskal) = \Theta(a \log n)$ --- $T(prim) = \Theta(n^2)$
Si $a \approx n$ conviene kruskal, si $a \approx n^2$ conviene prim

Si se usa un heap invertido para obtener el arco minimal en cada iteración. el orden del tiempo de ejecución de Prim cae a $\Theta(a \log n)$.
Y si uso lista de adyacencias?

Correctitud de ambos algoritmos:

Lema 25: sea G = (N,A) un grafo no dirigido, conexo, con pesos; $B \subset N$ y $T \subseteq A$ un conjunto promisorio de arcos tal que ninguno de sus miembros toca B. Luego si (u,v) es uno de los arcos minimales que tocan B, entonces $T \cup \{(u,v)\}$ también es promisorio.

Teorema: El algoritmo de Prim/Kruskal devuelve en T un árbol minimal de cubrimiento para todo G conexo

Prueba: Por inducción sobre i, se prueba que todo Ti es promisorio usando el lema 25, con el mismo B del algoritmo. Luego el Ti final es una solución optimal porque no puede tener más arcos.

Puentes

sea G no dirigido, conexo. Un puente es un arco de G cuya eliminación deja al grafo resultante disconexo. Un punto de articulación es un nodo de G cuya eliminación (junto con todos sus arcos incidentes) deja al grafo resultante disconexo.

Lema 28

Sea $G = \langle N, A \rangle$ un grafo no dirigido. Un arco $(u, v) \in A$ es un puente en G si y solo si (u, v) no pertenece a ningún ciclo simple en G.

Para saber si un arco es B (hacia atrás - produce ciclo)

(u, v) *es B si y solo si* $\mathtt{d}[v] < \mathtt{d}[u] < \mathtt{f}[u] < \mathtt{f}[v]$

Demostración.

Sea (u, v) un arco del grafo, y supongamos que u es descubierto antes que v. Luego v es descubierto y finalizado antes de finalizar u. Si el nodo v tiene como padre a u entonces (u, v) es un arco de la foresta; si el nodo v tiene otro padre, entonces (u, v) es un arco hacia atrás. En forma similar se prueba si v es descubierto antes que u. □

Utilizo un DFS. Cada vez que se encuentra un arco hacia atrás (que equivale a un ciclo simple por el lema 11), se marcan todos los arcos del ciclo como “no puentes”. Al finalizar el recorrido, los arcos que quedan sin marcar son puentes.
Si tengo un adyacente que está marcado como gris, significa que ya lo visité y tengo un ciclo, marco los arcos de ese ciclo como false, los que quedan en true son puentes.

Puntos de articulación

La raiz es punto de articulación si y solo si tiene al menos 2 hijos.
Un nodo n (no raiz) es punto de art si y solo si existe un hijo **u** de **n** tal que ningún descendiente v de u tiene un arco (v,w) donde w es ancestro propio de n.

De acuerdo al lema anterior, para saber si un nodo n es un punto de articulación es suficiente con verificar si se puede acceder a un ancestro propio desde un descendiente de n por un arco fuera de la foresta.
Teniendo en cuenta la numeración de descubrimiento generada por un recorrido DF, se puede computar para cada nodo el ancestro más viejo que se alcanza a través de **arcos hacia atrás B.**

$$\mathtt{masviejo}[v] = min \begin{cases} \mathtt{d}[v] \\ \mathtt{d}[w] : (v, w) \text{ es un arco hacia atrás} \\ \mathtt{masviejo}[w] : (v, w) \text{ es un arco de la foresta} \end{cases}$$

```
PROCEDURE PuntosArticulación(G)
    //masviejo : tiempo de descubrimiento del ancestro más viejo de i.
    array d[1..n], masviejo[1..n], pArt[1..n]
    DFS(G,d) //calculo tiempos de descubrimiento
    DFS(G,masviejo) //calcular masviejo
    pArt[1]::= tiene más de un hijo en la foresta? es la raiz
    FOR cada v en N-{1}
        pArt[v]::= existe un hijo u de v tal que d[v]<=masviejo[u]?
    RETURN pArt
```

el tiempo de ejecución es de **Θ(n + 2a) = Θ(a)** (dos recorridos más una iteración sobre todos los nodos) se podrían mejorar las constantes realizando el cálculo completo en un sólo recorrido
S(n) = Θ(n) arreglos d y masviejo, dfs

Ford-Fulkerson

Una red de flujo es un grafo dirigido G = (N,A) donde cada arco (u,v) ∈ A tiene asociado una capacidad c(u,v) ≥ 0, y se distinguen dos nodos s y t llamados fuente y destino, tal que ningún arco llega a la fuente, o sale del destino.

un flujo en G es una función $f : N \times N \rightarrow R^+$ que satisface:
restricción de capacidad: $0 \leq f(u,v) \leq c(u,v)$ para todo $u,v \in N$
conservación de flujo: $\sum v \in N - \{s,t\} f(u,v) = 0$

Problema: dada una red de flujo G, el problema MAXFLUJO consiste en encontrar el máximo flujo que admite G, esto es, la máxima capacidad de flujo que puede ingresar a través de la fuente y salir por el nodo de destino.

Se basa en los conceptos de red residual, incremento, camino de aumento y corte:

Every edge of a residual graph has a value called ***residual capacity*** which is equal to original capacity of the edge minus current flow.

Capacidad residual: es la capacidad adicional de flujo que un arco puede llevar: $c_f(u,v) = c(u,v) - f(u,v)$

Grafo residual de una red de flujo es un grafo que indica posible flujo adicional. Si hay un camino desde el origen hacia el destino en el grafo residual, entonces es posible agregar flujo.

Red residual: camino de la fuente al sumidero, donde cada uno de los arcos tiene un flujo residual mayor que cero. Siendo el flujo residual, el flujo que se puede obtener en un arco una vez que haya pasado un flujo por él.

Aumento de camino: se basa en ir aumentando el camino, hasta alcanzar el máximo (capacidad residual, definido anteriormente).

Corte en redes de flujo: consiste simplemente en realizar una partición del conjunto de vértices en dos subconjuntos.

Wikipedia()El procedimiento para obtener el flujo máximo de una red, consiste en seleccionar repetidas veces cualquier trayectoria de la fuente al destino y asignar el flujo máximo posible en esa trayectoria. Este algoritmo es un método iterativo, el cual, empieza con un flujo nulo y en cada iteración se va obteniendo un valor del flujo que va aumentando el camino, hasta que no se pueda aumentar más. se denomina método de Ford-Fulkerson, y en cada iteración encuentra un camino de aumento, obtiene el flujo de ese camino y se lo suma al flujo anterior, resultando en un flujo con valor mayor

Libro()
El método de Ford-Fulkerson aumenta iterativamente el valor del flujo. Comenzamos con f(u,v) = 0 para todo u,v en V , dando un flujo inicial de valor 0. En cada iteración, aumentamos el valor del flujo en G encontrando un "camino de aumento" en una "red residual" asociada **Gf** . Una vez que conocemos los arcos de un camino de aumento en Gf , podemos identificar fácilmente arcos específicos en G para los que podemos cambiar el flujo de modo que aumentemos el valor del flujo. Aunque cada iteración del método Ford-Fulkerson aumenta el valor del flujo, veremos que el flujo en cualquier arco concreto de G puede aumentar o disminuir; puede ser necesario disminuir el flujo en algunos arcos para que un algoritmo pueda enviar más flujo desde el origen al destino. Aumentamos repetidamente el flujo hasta que la red residual no tenga más caminos de aumento. El teorema del corte del flujo máximo demostrará que, al finalizar este proceso, se obtiene un flujo máximo.

```
FORD-FULKERSON-METHOD(G,s,t)
    Gf = Crear_grafo_residual(G);
    flujo[u,v]= 0
    mientras haya un camino de aumentación p en la red residual Gf
        aumentar el flujo f a lo largo de p
    return f
```

Si **f*** denota un flujo máximo en el grafo transformado, entonces una implementación directa de FORD-FULKERSON ejecuta el bucle while como máximo **|f*|** veces, ya que el valor del flujo aumenta al menos una unidad en cada iteración.
El tiempo para encontrar un camino en un grafo residual es, por tanto, **O(V+E') = O(E)** tanto si usamos DFS como BFS. Cada iteración del bucle while lleva, por tanto, un tiempo O(E), al igual que la inicialización en las líneas 1-2, lo que hace que el tiempo total de ejecución del algoritmo FORD-FULKERSON sea **O(E |f*|)**.

S(n) = O(grafo residual) = O(n^2) con matriz o **O(na)** con listas

Usos: modelar el tráfico en un sistema de autopistas, fluidos viajando en tuberías, corrientes eléctricas en circuitos eléctricos o sistemas similares por lo que viaje algo entre nodos, maximizar el flujo de paquetes por una red de computadoras.

El algoritmo de **Edmonds-Karp** es una instancia del método de Ford-Fulkerson en donde la búsqueda del camino de aumento **p** se hace mediante una búsqueda por niveles BFS comenzando por **s**.
El recorrido por niveles permite encontrar los caminos mínimos (en cuanto a cantidad de arcos) desde el origen a cada nodo.
Entonces en la red residual, el camino mínimo ya no existe. Esto permite ajustar la cantidad de iteraciones del ciclo WHILE.
El algoritmo EK en G = (N,A) toma O(na) iteraciones. como cada iteración (construir el grafo residual, hacer BFS, encontrar el camino de aumento y la capacidad residual, y actualizar el flujo) del algoritmo

EK toma de O(a), de acuerdo al teorema anterior el tiempo total de ejecución es de **$O(na^2)$**. Se elimina de esta forma la dependencia del tiempo de ejecución en **f***

Análisis amortizado

El análisis amortizado estudia el tiempo requerido para ejecutar una secuencia de operaciones sobre una estructura de datos. Si usamos el análisis normal en el peor caso, ejecutar N operaciones sobre una estructura de datos de n elementos lleva tiempo en O(Nf(n)), donde f(n) es el tiempo en el peor caso de la operación. En muchos casos esa cota no es ajustada debido a que el peor caso puede NO ocurrir las N veces, o incluso ninguna de esas veces.
Entonces se introducen las técnicas de análisis amortizado para tratar de obtener una cota menor para la serie de operaciones

El análisis amortizado se diferencia del análisis en el caso promedio en que no involucra probabilidades, y en que garantiza el tiempo en el peor caso de las N operaciones. El análisis probabilístico produce un tiempo esperado que una determinada ejecución puede sobrepasar o no. El análisis amortizado produce una cota en el tiempo de ejecución de la serie de operaciones (es decir, sigue siendo un análisis del peor de los casos).

Tenemos operaciones que realizan tareas en función de las próximas operaciones que serán llamadas, para hacerlas más eficientes. Ej conjuntos disjuntos - Una secuencia de N llamadas es O(N log* n), no es una cota individual para cada llamada, es una cota para el conjunto de las llamadas, puede haber algunas que sean ineficientes y puede haber otras que sean muy eficientes.
El análisis amortizado es como que balancea los costos entre las eficientes y las no eficientes.

La cota del peor de los casos es muy grande xq se da muy poco el peor caso.

En conjuntos disjuntos la eficiencia estaba dada por la compresión de caminos. Esa operación no es necesaria pero si sirve para las próximas ejecuciones.

Varios métodos para realizar el análisis amortizado. Usamos el **método del potencial.**
Este método representa los ahorros hechos en operaciones previas mediante incrementos en la "energía potencial", que puede ser gastada en operaciones costosas siguientes. La energía potencial representa las tareas realizadas en llamadas anteriores de una operación, que pueden ser usadas en próximas invocaciones a la operación. Se comienza con una estructura de datos inicial D_0, y para cada i= 1,2,...,N sea c_i el costo real de la i-ésima operación y D_i la estructura de datos resultante después de aplicar esa operación sobre D_{i-1}.

La función phi(D_i) es el valor de la función potencial en el estado en el momento i de la estructura de datos.
Hay funciones que aumentan el potencial(el ahorro), son funciones baratas.
El costo amortizado (para cu de las N operaciones) de la i-ésima operación se define como:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

C_i (costo real) + la diferencia de potencial después de ejecutar la operación.
Una operación con costo c_i alto, se puede ver compensada por el potencial positivo de operaciones baratas anteriores.

La sumatoria de los costos amortizados de la secuencia de operaciones es:

$$\sum_{i=1}^{N} \hat{c}_i = \sum_{i=1}^{N} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{N} c_i + \Phi(D_N) - \Phi(D_0)$$

Si se cumple una serie de condiciones la sumatoria de los costos amortizados es una cota superior de los costos individuales c_i. La sumatoria de los costos reales está acotada superiormente por la sumatoria de los costos amortizados.
Si se puede definir Φ tal que $\Phi(DN) \geq \Phi(D0)$ entonces este valor es una cota superior a la sumatoria de costos reales.
Si la función está bien elegida y la E.D. está diseñada para ello, esta cota superior es más ajustada que N veces el peor caso.

El costo real nunca puede ser negativo. El costo amortizado sí.

Pila extendida

consideremos una E.D. Pila con la operación adicional DesapilarMúltiple(k) que elimina los k elementos que están más arriba en la pila. Las operaciones Apilar, Desapilar y Tope son de Θ(1) en el peor caso individual, pero la operación DesapilarMúltiple(k) es de Θ(min(k,n)), siendo n los elementos en la pila. Esto da un tiempo de O(nN) para cualquier secuencia de N operaciones.

Tomemos como función potencial Φ(Di) la cantidad de elementos en la pila Di.
sea s la cantidad de elementos después de i −1 operaciones

Apilar: $\hat{c}i = ci + \Phi(D_i) - \Phi(D_{i-1}) = 1 + s + 1 - s = 2 \in \Theta(1)$

Tope: $\hat{c}i = ci + \Phi(D_i) - \Phi(D_{i-1}) = 1 + s - s = 1 \in \Theta(1)$

Desapilar: $\hat{c}i = ci + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (s-1) - s = 0 \in \Theta(1)$

DesapilarMúltiple(k), con $k \leq s$: $\hat{c}i = ci + \Phi(D_i) - \Phi(D_{i-1}) = k + s - k - s = 0 \in \Theta(1)$
DesapilarMúltiple(k), con $k > s$: $\hat{c}i = ci + \Phi(D_i) - \Phi(D_{i-1}) = s + 0 - s = 0 \in \Theta(1)$

Este análisis amortizado permite afirmar que N operaciones sobre esta E.D. lleva tiempo en Θ(N) en el peor caso.

Contador binario

Se tiene un contador binario de n bits con la operación de incremento. El costo de esta operación es la cantidad de bits cambiados. En el peor caso, la cantidad de bits cambiados son todos los bits del contador, por lo que N incrementos es de O(nN). Es claro que el peor de los casos no se da en todas las N operaciones, por lo que es válido realizar un análisis amortizado.

Para hacer un análisis amortizado de esta E.D. tomaremos como función potencial la cantidad de 1's en el contador. Si empezamos con el contador en cero, $\Phi(Di) \geq \Phi(D0)$ para todo i, por lo que podemos asegurar que la sumatoria de costos amortizados es una cota superior a la sumatoria de costos reales.

Sea s la cantidad de 1's del contador antes del incremento, t de los cuales se modifican en la i-ésima operación, el costo amortizado de la operación es: $\hat{c}i = ci + \Phi(Di) - \Phi(Di-1) = (1+t) + (s - t + 1) - (s) = 2 \in \Theta(1)$. Luego, N operaciones de incremento del contador llevan en el peor caso tiempo de Θ(N)

Skew heaps

un heap asimétrico (o skew heap) es un árbol binario que respeta la propiedad de heap, no tiene restricciones estructurales, y soporta las operaciones **inserción**, **eliminarMin** y **mezcla**. **inserción** y **eliminarMinimo** se implementan en base a **mezclar**.

insertar(elem, heap): mezcla un árbol con un único elemento **elem** con un árbol con más elementos.

eliminarMin(heap): elimino la raíz del árbol y me quedan 2 árboles sueltos. Mezclo estos 2 árboles.

Mezclar() en un heap original nos rompe todo lo bueno de las operaciones de la estructura. El T(n) es de O(n) xq se crea un nuevo árbol con los dos árboles a mezclar.

Los heaps asimétricos tienen un tiempo amortizado de **logn.** Una secuencia de N operaciones sobre esta estructura es del orden de **Nlogn**.

```
PROCEDURE mezcla(T1,T2)
    IF (T1 == null and T2 == null)
        RETURN null
    IF (T2 == null) or (T1 != null and T1.raiz() < T2.raiz())
        this.setRaiz(T1.raiz())
        this.setHD(T1.hi())
        this.setHI(T1.mezcla(T2, T1.hd())
    ELSE
        this.setRaiz(T2.raiz())
        this.setHD(T2.hi())
        this.setHI(T2.mezcla(T1,T2.hd()))
    RETURN this
```

Agarro la raíz más chica entre los dos heaps y esa es la raíz del nuevo árbol.
A medida que voy navegando el subárbol derecho y voy mergeando, el resultado lo swapeo. Lo que era camino derecho lo pongo como camino izquierdo.
To merge two skew heaps, we merge their right paths and make this the new left path. For each node on the new path, except the last, the old left subtree is attached as the right subtree
Mezclo los dos caminos derechos de los heaps y ese es el camino izquierdo del nuevo heap

El tiempo de ejecución es proporcional a la cantidad de elementos que tengo en los caminos derechos de ambos heaps a mezclar. En el peor caso tengo 2 heaps donde todos los elementos están en el camino derecho, entonces el T(n) es de O(n). Pero como estos nodos pasan a continuación a estar en el camino izquierdo, las próximas mezclas no los considerarán, esto amerita realizar un análisis amortizado de esta operación

Se toma como **función potencial** a la cantidad de **nodos pesados**: aquellos nodos que tienen al menos la mitad de sus descendientes en el subárbol derecho. Los otros nodos son **livianos.**
La cantidad de nodos del camino derecho determina el tiempo de la operación mezcla acotar estos nodos nos permite determinar los tiempos amortizados
La cantidad de nodos **livianos** en el camino derecho es de O(log n) xq la mayor cantidad de sus descendientes están en la parte izquierda.

Sea p1, p2, l1, l2 la cantidad de nodos pesados y livianos del camino derecho de T1 y T2.
El costo real de la mezcla es p1+l1+p2+l2 (algunos pesados y otros livianos).
La diferencia de potencial:

- Los nodos **pesados fuera del camino** derecho de T1 y T2 no se transforman, siguen siendo pesados después de la mezcla. Se cancelan en la resta de potenciales.
- Los nodos **pesados del camino** derecho van a ser girados y pasan a ser livianos. Entonces restan en t3 (desaparecen).
- Los nodos **livianos balanceados del camino** derecho, siguen siendo balanceados = livianos. Se cancelan.
- Los nodos **livianos no balanceados del camino** derecho, al ser girados pasan a ser pesados.

Entonces la diferencia de potencial queda -p1-p2+l1+l2. Suponiendo que todos los livianos pasan a ser pesados me queda una cota superior

El potencial en el momento i-1 va a ser el potencial de la suma entre t1 y t2. Por lo que nos queda:

$$\hat{c}_i = p_1 + l_1 + p_2 + l_2 + \Phi(D_i) - \Phi(D_{i-1})$$

$$\begin{aligned} \hat{c}_i &= p_1 + l_1 + p_2 + l_2 + \Phi(T_3) - \Phi(T_1, T_2) \\ &\leq p_1 + l_1 + p_2 + l_2 - p_1 - p_2 + l_1 + l_2 \\ &= 2(l_1 + l_2) \in O(\log n_1 + \log n_2) = O(\log n) \end{aligned}$$

Como vimos antes por teorema 1, la cantidad de nodos livianos en el camino derecho es a lo sumo logn. Por lo que el costo amortizado es **O(log n).** ⇒ Se prueba por
inducción

Esto implica que una serie de **N** operaciones sobre skew heaps, empezando de la estructura vacía, toman tiempo de **O(N log n)** en el peor caso

Usos: Algoritmo de prim. Para encontrar los árboles de cubrimiento minimal, buscaba el nodo más cercano entre todos los que quedaban. Una vez que lo incluiamos en la solución parcial que teníamos, tenemos que mezclar todos los nodos más cercanos a este nodo con todos los nodos más cercanos que teníamos de los otros nodos encontrados en la solución parcial. esto se puede hacer a través de que cada tnodo tenga un heap, este heap nos dice la adyacencia de ese nodo, nos va a dar cual es el más cercano a ese nodo, cuando incluimos ese nodo a la solución parcial hacemos la mezcla con los de la solución parcial anterior y los más cercanos de este.mezclar todos los nodos con menores buscar el nodo más cercano entre todos los que quedaban.

Fibonacci Heaps

Su estructura se basa en los heaps binomiales. Un heap binomial es una **foresta/colección** de árboles binomiales que cumple con las siguientes propiedades:

cada árbol de la foresta cumple con la propiedad de heap.

existe en la foresta a lo sumo un árbol binomial de cada rango

un árbol binomial de rango k, notado Bk , se define inductivamente como: B0 tiene un sólo nodo, y Bk se forma enlazando dos árboles binomiales de rango k −1 de manera que uno sea hijo del otro

Sea Bk el árbol binomial de grado k, entonces:

1. Bk tiene 2^k nodos
2. Bk tiene altura k
3. Bk tiene k hijos

Dado que la mezcla de dos árboles binomiales lleva un tiempo constante (pongo a uno como hijo del de menor raíz), y hay O(logN) árboles binomiales, la mezcla lleva un tiempo O(logN) en el peor de los casos.

OPERACIONES DE HEAPS BINOMIALES:

- **mínimo**() puntero al árbol con menor clave en tiempo **Θ(1)**
- **mezclar**(H1, H2) compone árboles binomiales de rangos repetidos en un árbol de rango mayor. Hay O(logN) árboles binomiales, la mezcla lleva un tiempo **O(logN)**
- **eliminarMinimo**() e **insertar**(x) se implementan en base a la operación de mezcla
- **disminuirClave**(x, k) debe recorrer a lo sumo la máxima altura del árbol más alto en la foresta, lo que es de O(log n) de acuerdo a las propiedades vistas.
- **eliminar**(x) se implementa disminuyendo la clave del elemento al mínimo posible, y luego llamando a **eliminarMinimo**().

Los heaps de Fibonacci se diferencian en que a medida que se ejecutan las operaciones, no necesariamente siempre estos árboles mantienen su estructura binomial

OPERACIONES DE HEAPS DE FIBONACCI: Se hacen estas modificaciones

- **mezcla perezosa**: Posterguemos la mezcla hasta que sea necesario(al eliminar min). Dos heaps se mezclan simplemente concatenando las forestas y calcula el nuevo mínimo **Θ(1)**. No siempre existe un único árbol de cada rango. Favorece el tiempo de mezcla e inserción, pero aumenta el de eliminarMinimo.
- **cortes** para mejorar el tiempo del burbujeo en la implementación de disminuirClave. Entonces cuando un nodo tiene clave menor que el padre, se elimina cortando el subárbol y agregándolo como un árbol nuevo a la foresta. Favorece el disminuirClave, pero perjudica a eliminarMinimo. Esto hace que los árboles vayan perdiendo altura y queden nodos desparramados.
- **cortes en cascada** si un padre ha perdido más de un hijo, lo que asegura mantener la cantidad de descendientes de todos los nodos. Cada vez que un padre pierde un hijo, lo marcamos, si tiene que perder un 2do hijo, le ejecutamos desp corte al padre.

La operación de **EliminarMinimo** es la encargada de restaurar la propiedad de que la foresta sea una colección de árboles donde existe a lo sumo uno de cada rango. La implementación utiliza un procedimiento auxiliar consolidar() que controlan que para cada árbol no existe otro de su rango. Si esto sucede, los mezcla y crea un árbol de rango superior **O(n)**

```
PROCEDURE EliminarMinimo()
    z::=this.minimo()
    IF z != null
        FOR cada hijo x de z
            agregar x a la foresta
        eliminar z de la foresta
        this.consolidar()
    RETURN z
```

La operación **disminuirClave** debe implementarse en tiempo amortizado constante, por lo que no es posible hacer un burbujeo por el árbol.
Si cuando se disminuye la clave se viola la propiedad de heap, entonces el nodo es cortado del árbol al que pertenece y se inserta como un árbol independiente en la foresta.
Para asegurar que un nodo no pierda demasiados descendientes (y por lo tanto asegurar el tiempo amortizado del eliminarMinimo), entonces se marca el nodo que pierde un hijo por primera vez
Si un nodo pierde un segundo hijo, entonces también es cortado, se agrega su subárbol como árbol independiente en la foresta, y se procede con el padre

una llamada a DisminuirClave tiene como costo real 1 más la cantidad de cortes en cascada

Análisis amortizado:

Tengo una función costosa eliminarMinimo() que en el peor de los casos toma **O(n).** Elimino el mínimo y todos sus hijos los pongo como árboles de la foresta. En el peor caso tengo todos árboles de 1 nodo. Después de ejecutar eliminar minimo, tengo a lo sumo 1 árbol de cada rango, y para volver a ser O(n) se tiene que degenerar todo de vuelta.
Función potencial: cantidad de árboles + 2 * cantidad de nodos marcados
cantidad de árboles me permite ahorrar en eliminar minimo y la otra ahorrar en los cortes en cascada

$$\Phi(H) = arboles(H) + 2 \times marcados(H)$$

insertar(): se agrega un árbol y los nodos marcados no cambian: **Θ(1) + 1 = Θ(1)**

mezcla(): la cantidad de árboles y nodos marcados no cambian: **Θ(1)**

disminuirClave(): el costo real es 1 más la cantidad de cortes en cascada C(padre, abuelo, etc).

Sea M la cantidad de nodos marcados antes de la operación. La cantidad de árboles aumenta en 1+C, C nodos marcados dejan de serlo, y un nodo no marcado pasa a marcado:

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(H_i) - \Phi(H_{i-1}) \\ &\leq 1 + C + (T + 1 + C + 2(M - C + 1)) - (T + 2M) = 4 \in \Theta(1) \end{aligned}$$

2(M-C+1) los C nodos cortados ya no están marcados.

eliminarMinimo(): para esta operación debo acotar el rango/grado de los nodos - esta cota es la que le da el nombre de fibonacci - el rango/grado de un nodo me da una cota en la cantidad de descendientes de ese nodo - grado k tengo 2^k descendientes en binomiales - En los fibonacci heap, un árbol de grado **k** va a tener (número de oro)^k nodos)

Esta implementación(con mezcla perezosa, cortes, cortes en cascada) hacen que se de esta propiedad.

Lema 4

Sea x un nodo en un heap *de Fibonacci tal que* $\texttt{grado}[x] = k$.
Entonces para todo hijo $y_i, 2 \leq i \leq k$ vale que $\texttt{grado}[y_i] \geq i - 2$.

Teorema 6

Sea x un nodo de un heap *de Fibonacci tal que* $\texttt{grado}[x] = k$.
Encontes la cantidad de descendientes de x es al menos F_{k+2}.

Demostración.

Por inducción sobre el rango k de x. Si $k = 0$ vale. Si $k > 1$, sean $y_1, \ldots, y_k$ los hijos de x en el orden de inserción. Entonces

$$des(x) \quad = \quad 1 + 1 + \sum_{i=2}^{k} des(y_i) \geq 1 + 1 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = 1 + F_{k+2} - 1 = F_{k+2}$$

Supongamos que tenemos T árboles antes de ejecutar eliminarMin(), eliminamos un árbol pero sumamos r nuevos árboles, que son los r hijos que tenía el nodo min. Y después llama a consolidar de O(n).

El costo real de la operación es **T-1+r.** Consolidamos los T árboles que teníamos, menos el que elimine min más los r nuevos árboles.

eliminarMin() no cambia la cantidad de nodos marcados, se cancela en la diferencia de potencial.

Sí cambia la cantidad de árboles, en **i-1** tengo T árboles y en **i** voy a tener a lo sumo 1 árbol de cada rango, a lo sumo (log n) árboles(teorema).

$$\Phi(H) = arboles(H) + 2 \times marcados(H)$$

$$\hat{c}_i \quad = \quad c_i + \Phi(H_i) - \Phi(H_{i-1}) = T\text{-}1 + r + O(\log n) - T$$

El rango es O(log n) teo 6.

$$\leq \quad r + O(\log n) \leq O(\log n) + O(\log n) \in O(\log n)$$

USOS: su utilización sería útil en algoritmos como el de Dijkstra para los caminos más cortos con orígen único, donde en cada iteración de ciclo greedy no sólo se necesita seleccionar el nodo más próximo en el heap, sino también disminuir la distancia de los restantes nodos

Complejidad computacional

Objetivos:
el concepto de correctitud de un algoritmo debe no sólo incluir el hecho de que el algoritmo realice lo especificado, sino también que lo realice con los recursos disponibles.
El objetivo fundamental de la Complejidad Computacional es clasificar los problemas de acuerdo a su tratabilidad, tomando el o los algoritmos más eficientes para resolverlos.

Clasificación de problemas:

Un problema es **tratable** si tiene solución para instancias grandes, son problemas decidibles que tienen algoritmos eficientes con cantidad polinomial de tiempo y espacio.
En cambio, es **intratable** si solo se pueden resolver instancias pequeñas y consumen demasiados recursos, tales como el problema de las torres de Hanoi, monkey puzzle, ajedrez.

Conceptos básicos:

con el objetivo de simplificar la clasificación, sólo se considerarán problemas de decisión. Cualquier problema, de optimización, etc, se puede resolver usando problemas de decisión.

Se establecerá una **asociación entre problemas de decisión y lenguajes formales** y **se clasificarán lenguajes formales**, en lugar de problemas de decisión.

Se usará como modelo de computación a las máquinas de Turing.

Sea T una MTD, y NT una MTND. La computación T(x) se dice que acepta a x si T(x) es finita y termina en el estado aceptador q1. La computación NT(x) se dice que acepta a x si existe en el árbol T(x) un camino finito cuya hoja está en el estado aceptador q1.
El **conjunto de entradas que T (o NT) acepta** se denomina el **lenguaje aceptado** por T (o NT), y se nota **L(T)** (o L(NT)).

Teorema: Sea NT una MTND. Siempre existe una MTD T tal que L(NT) = L(T) y si x es aceptada por NT en t pasos, entonces x es aceptada por T en c^t pasos, para alguna constante positiva c (cantidad de estados).

Un lenguaje $L \subseteq \Sigma^*$ es **aceptable** (resuelve las instancias sí, no me interesa las de no) si existe una MTD T tal que L = L(T).
Un lenguaje $L \subseteq \Sigma^*$ es **decidible** (termina en todos los casos) si existe una MTD T tal que L = L(T) y además, para todo $x \in \Sigma^*$ T(x) es finita.

Si un lenguaje L es decidible entonces L es aceptable

Para comparar lenguajes, se usarán **funciones de reducibilidad**: dados dos lenguajes $A,B \subseteq \Sigma^*$, una función computable $f : A \rightarrow B$ es una **función de reducibilidad** si para todo $x \in \Sigma^*$ vale $x \in A \leftrightarrow f(x) \in B$.
La función de reducibilidad es una transformación de instancias de un problema en instancias de otro problema tal que la solución al segundo es la solución al primero (una reducción entre problemas)
Se dice que $A \leq_m B$ si existe una función de reducibilidad de A hacia B.

Si L1 ≤m L2 y L1 no es decidible, entonces L2 no es decidible.

Clases de complejidad:

El estudio de la complejidad clasificará los lenguajes decidibles de acuerdo a la cantidad de recursos necesarios para su computación.

Los problemas **tratables** son problemas decidibles que tienen algoritmos eficientes con cantidad polinomial de tiempo y espacio. Los **intratables** consumen demasiados recursos.

Una clase de lenguaje es un conjunto de lenguajes que cumple con alguna propiedad, es decir C = {L : L $\subseteq$ Σ^* y π(L)}.
Dadas dos clases de lenguajes C1 y C2, si se quiere mostrar que C1 $\subseteq$ C2 entonces es suficiente con probar que para todo L $\subseteq$ Σ^* si vale π1(L) entonces también vale π2(L).
Si además de C1 $\subseteq$ C2 se quiere mostrar que la inclusión es estricta (C1 $\subset$ C2), entonces se recurre a buscar un lenguaje separador que pertenezca a C2 pero no a C1.

Lema 8: Sean C1 $\subset$ C2 dos clases de complejidad tal que C1 es cerrada c.r. a $\leq_r$. Entonces cualquier L $\in$ C2 que sea C2-completo c.r. a $\leq_r$ es tal que L $\notin$ C1.

Una clase C es cerrada c.r. a $\leq_r$ si para todo L1 $\in$ C vale que L2 $\leq_r$ L1 implica L2 $\in$ C . Un lenguaje L1 $\in$ C es completo c.r. a $\leq_r$ si para todo L2 $\in$ C vale que L2 $\leq_r$ L1

Clase P:

Contiene a (casi) todos los problemas tratables, aquellos que pueden ser **decididos** por una **MTD** en **tiempo polinomial.**
la mayoría de los problemas de esta clase tienen algoritmos con implementación razonable, a pesar de que el algoritmo polinomial para ese problema no sea el más conveniente de usar en la práctica.

Hay ciertos problemas que no son realmente tratables debido a que el grado del polinomio o las constantes ocultas son muy grandes pero no se tienen en cuenta y se considera a P una aproximación razonable a los problemas tratables.

Es posible demostrar que un **problema pertenece a P** mediante dos formas:

- mostrando un algoritmo polinomial determinístico
- usando una reducción determinista polinomial a otro problema que ya se sabe que está en P

Reducibilidad polinomial: dados dos lenguajes L1 y L2, L1 se dice polinomialmente reducible a L2 si existe un función f computable en tiempo polinomial tal que x $\in$ L1 $\leftrightarrow$ f(x) $\in$ L2. Se nota L1 $\leq_p$ L2.

La reducción polinomial transforma en **tiempo polinomial** una instancia del problema L1 en una instancia del problema L2, de forma que sus respuestas sean las mismas.

Además, permite afirmar que P es cerrada c.r. a $\leq_p$: Sean L1,L2 dos lenguajes tales que L1 $\leq_p$ L2. Luego si L2 $\in$ P entonces L1 $\in$ P.

Clase NP:

Es el conjunto de problemas que puede ser resuelto en tiempo no determinístico polinomial, es decir, aquellos que pueden ser **dicididos** por una **MTND** en **tiempo polinomial.**

Es posible demostrar que un **problema pertenece a NP** mediante dos formas:

- Dando una MTND que resuelva el problema en tiempo polinomial
- Mostrar que su lenguaje L_{check} pertenece a P. Esto es, se da un algoritmo determinístico polinomial para chequear la solución dada por el algoritmo.

Por definición (toda computación determinística es trivialmente no determinística), sabemos que **P ⊆ NP.** Pero el problema de definir la "tratabilidad" no es tan sencillo: no se sabe si P = NP o P ⊂ NP. Esto es, puede ser que todos los problemas en NP tengan solución determinística polinomial, o puede ser que no sea equivalente el tiempo determinístico con el no determinístico.

Hay problemas muy útiles como el de la mochila con coeficientes enteros, el del viajante y factorizar un número primo que no se sabe dónde caen.
Una característica interesante es que todos estos problemas parecen requerir, como parte de su naturaleza, construir soluciones parciales. Cuando una de estas soluciones parciales se detecta incorrecta, se debe realizar backtracking en busca de otra alternativa por lo tanto es difícil decidir si existe o no una solución; es decir dar la respuesta sí o no.
En cambio es interesante notar que en el caso que la respuesta sea sí es fácil convencer a alguien de ello. Esto es lo que se denomina certificado, cuyo tamaño siempre puede ser acotado por Θ(n).

Ejemplos de NP:

- **SAT**. dar un algoritmo determinístico polinomial para chequear si una asignación de valores de verdad hace verdadera una fbf
- **3-COLOR.** dando un algoritmo polinomial para controlar que una asignación de colores de un grafo asigne colores diferentes a nodos adyacentes.
- **Mochila con coeficientes enteros**. Controlar es que alguna de todas las cargas generadas tenga peso igual al peso máximo, y eso se puede hacer en tiempo polinomial fácilmente. Nadie pudo generar un algoritmo P todavía.

Clase NPC:

Un problema C es NPC si:

1. C está en NPC y
2. Todo problema de NP es reducible a C en tiempo polinomial

La clase NPC es un subconjunto de problemas de NP tal que todo problema de NP se puede reducir polinomicamente a cada uno de los problemas NPC.

Como P es cerrada respecto a ≤p, permite suponer que los lenguajes **NP-completos** respecto a ≤p (NPC) son **candidatos a lenguajes separadores**, es decir a pertenecer a NP−P. En el caso que P != NP. (clave)
Entonces, para averiguar si P = NP es suficiente con establecer el status de un problema en NPC:

- Si uno de estos problemas tiene solución polinomial, entonces todos los NP tienen solución polinomial y **P = NP**
- Si se demuestra que no puede existir una solución polinomial para alguno de ellos, entonces ninguno de NPC la tiene. Entonces estos problemas (que son NP) ya no son P. Luego **P != NP**.

El destino de un problema NPC es el destino de todos: todos son tratables o todos son intratables. Por lo que es interesante conocer cuáles problemas pertenecen a NPC.

Teorema de cook: Establece que SAT es NPC

Simula el funcionamiento de una MTND, traduce su funcionamiento a una formula del calculo proposicional.
Esta fórmula es satisfacible sssi la computación de la MTND termina en tiempo polinomial.

La importancia del teorema de Cook radica en que a partir del conocimiento de que SAT$\in$ NPC, es mucho más fácil probar que otros problemas son NPC a través de la reducción polinomial:

L $\in$ NPC si y solo si L2 $\in$ NPC, L $\in$ NP y L2 $\leq_p$ L.

De esta forma más de 1000 problemas de dominios muy diferentes, y con variadas aplicaciones, se han probado que pertenecen a NPC. El resultado es muy frustrante, ya que para ninguno de estos problemas se ha podido encontrar una solución polinomial, y tampoco se les ha podido demostrar una cota inferior mayor que lineal.

Para probar que un problema X es NPC, demuestro que SAT se reduce a ese problema. Como yo sé que SAT es NPC (por teorema de Cook), entonces todos los problemas de NP se reducen polinomialmente a SAT, y como la reducción es transitiva, todos los demás problemas se van a reducir a mi problema X que quiero demostrar. Por lo tanto, X es NPC.
La clave es elegir un problema y hacer la reducción

Con demostrar que un lenguaje NPC está en P, ya estoy demostrando que P = NP. Si demuestro que no está en P, entonces ese lenguaje es separador y P != NP.

Si se prueba que P = NP podemos obtener un algoritmo polinomial para factorizar primo y atacar la criptografía basada en exp mod.

Ejemplos de NPC: SAT, viajero, ciclo hamiltoniano

Problemas intratables: Torres de Hanoi, Monkey Puzzle, Ajedrez

Algoritmos de aproximación:

El hecho de conocer que un problema es NP-completo es instructivo.
Puede ser útil para no perder tiempo en búsqueda de un algoritmo eficiente que probablemente no exista. Sin embargo, esto no significa que el problema no se pueda solucionar.

Un algoritmo heurístico, o simplemente heurística, es un procedimiento que eventualmente puede producir una solución buena, o incluso optimal, a un problema; pero también puede producir una mala solución o incluso ninguna. La heurística puede ser determinística o probabilística. La principal diferencia con los algoritmos Monte Carlo es que no está garantizada una baja probabilidad de error. Un algoritmo de aproximación es un procedimiento que siempre resulta en una solución al problema, aunque puede fallar en encontrar la solución optimal. Para que sea útil, debe ser posible calcular una cota o sobre la diferencia, o sobre el cociente entre la solución optimal y la devuelta por el algoritmo.

Otros algoritmos

InsertionSort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

```
FOR j ::= 2 TO n
    x ::= A[j]
    i ::= j-1
    WHILE i> 0 and A[i]>x
        A[i+1] ::= A[i]
        i ::= i-1
    A[i+1] ::= x
```

T(n) = O(n^2) --- S(n) = O(1)

Fibonacci

Naive: T(n) = T(n-1) + T(n-2) + b = **O(2^n)** (ecuación caract) --- **S(n) = O(n)** árbol de recursión

Iterativo: T(n) = b + $\sum_{k=1}^{n}$ (c1 +c2 +c3) +d = **Θ(n)** --- **S(n) = O(1)**

Iterativo complejo: T(n) = c1+ $\sum_{k=1}^{logn}$ c2+ $\sum_{k=1}^{logn}$ c3 = **O(logn)** --- **S(n) = O(1)**

HeapSort

```
FUNCTION Heapsort(A)
    Construir Heap(A)    //O(n)
    FOR i ::= n DOWNTO 2
        A[1] <=> A[i]
        A.tamaño--
        A.heapify(1)    //O(logn)
ENDFOR
```

$$T_H(n) = \Theta(n) + \sum_{i=2}^{n}(c_1 + c_2 + c_3 + \Theta(\log n)) = \Theta(n) + \Theta(n\log n) \in \Theta(n\log n)$$

SelectionSort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array. In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

```
FOR i ::= 1 TO n-1
    ind ::= i; min ::= A[i]
    FOR j ::= i+1 TO n
        IF (A[j]<min)
            min ::= A[j]
            ind ::= j
    A[ind] ::= A[i]; A[i] ::= min
```

$$T_S(n) = a + \sum_{i=1}^{n-1}(a+b+(n-i)c) \in \Theta(n^2)$$

S(n) = O(1)

Algoritmo de Euclides

Encuentra el MCD entre dos números m y n

```
Function EUCLIDES(m,n)
    WHILE m>0
        temp ::= m
        m ::= n mod m
        n ::= temp
    RETURN n
```

se puede observar las propiedades:

$n_i = m_{i-1}$, $m_i = n_{i-1} \bmod m_{i-1}$ siempre que $i \geq 1$
$n_i \geq mi$ siempre que $i > 1$
para todo n,m tal que $n \geq m$ vale $n \bmod m < n/2$
$n_i = m_{i-1} = n_{i-2} \bmod m_{i-2} < n_{i-2}/2$ si $i > 2$

Luego, en dos iteraciones n0 se reduce a menos de la mitad; en cuatro a menos del cuarto; etc. Como $m_i > 0$ entonces no puede haber más de $2\log_2(n_0)$ iteraciones. Y **T(n) ∈ O(log n).**

Euclides **lineal en la LONGITUD** de los datos de entrada. **Logarítmico en el VALOR.**
Cuando los datos de entrada son números, siempre se analiza la longitud de los números, la cantidad de dígitos.

S(n) = O(1)

Ordenamiento por cubículos

Ordena un arreglo de enteros con un rango limitado conocido. Enteros hasta **s.**

```
cubículos(T)
    array U[1..s] ::= 0    //O(n)

    FOR i ::= 1 TO n
        k ::= T[i]; U[k]++
    i ::= 0
    FOR k ::= 1 TO s
        WHILE U[k] != 0                  barómetro
            T[i++] ::= k; U[k]--
```

el barómetro se ejecuta $U[k]_0 + 1$ veces por cada k. Luego el tiempo total es: $\sum_{k=1}^{s} (U[k]_0 + 1)\Theta(1)$

$$T(n) = \Theta(n) + \Theta(1) + \sum_{k=1}^{s}(U[k]_0 + 1)\Theta(1) = \in \Theta(\max(n,s))$$

S(n) = Θ(max(n,s))

el problema de este algoritmo es el límite máximo de los números a utilizar, y el espacio de memoria auxiliar

Funciones de crecimiento suave

Sirven para extender lo analizado condicionalmente a todos los tamaños de entrada

Teorema 1: Sea $f : N \rightarrow R^+$ una función de **crecimiento suave**, y $t : N \rightarrow R^+$ una función **eventualmente no decreciente**. Luego siempre que $t(n) \in \Theta(f(n) \mid n = b^k)$ para algún entero $b \geq 2$, entonces $t(n) \in \Theta(f(n))$

Una función $f(n) : N \rightarrow R^+$ es **eventualmente no decreciente** si existe $n0 \in N$ tal que para todo $n \geq n0$ vale $f(n) \leq f(n+1)$.

Una función $f(n) : N \rightarrow R^+$ es de **crecimiento suave** si existe $b \in N, b \geq 2$ tal que f(n) es eventualmente no decreciente y $f(bn) \in O(f(n))$

la mayoría de las funciones que se encuentran son de crecimiento suave: log n, n, n log n, n 2 , o cualquier polinomio con coeficiente principal positivo

Investigación sobre Disjoint-Set

Resumen: Un disjoint-set es una estructura de datos que mantiene un conjunto S de elementos divididos en varios subconjuntos disjuntos S1, S2, ..., Sn de forma tal que en todo momento cada objeto pertenece a exactamente un único conjunto. Cada subconjunto es identificado por un "representante", que es algún elemento de dicho subconjunto.

Operaciones básicas:

find(x): retorna el elemento representante del único conjunto que contiene a x. Sigue la cadena de padres en el árbol desde x hasta que alcanza un elemento raíz. Este elemento raíz es el representante del conjunto, y puede ser el propio x.

merge(x,y): une los conjuntos disjuntos que contienen a x e y, llámense Sx y Sy, en un nuevo conjunto. El representante de este nuevo conjunto será algún miembro de Sx U Sy aunque generalmente se elige a alguno de los dos antiguos representantes de Sx o Sy.

Tiempos:

find(x) debe recorrer todos los padres desde x a la raíz. En el peor caso, se forma una lista de padres por lo que para obtener el padre de x debemos recorrer los n elementos requiriendo O(n).

merge(x,y) usa Find para determinar las raíces de los árboles a los que pertenecen x e y. Si las raíces son distintas, los árboles se combinan uniendo la raíz de uno con la del otro. Si esto se hace de manera ingenua, como por ejemplo haciendo siempre que x sea hijo de y, la altura de los árboles puede crecer a O(n). Para evitar esto, podemos utilizar heurísticas como union by rank. Con esta heurística, siempre se une el árbol más corto a la raíz del árbol más alto. Así, el árbol resultante no es más alto que los originales, a menos que éstos tuvieran la misma altura, en cuyo caso el árbol resultante es más alto por un nodo.

Otra heurística para mejorar el desempeño de find(x) es la compresión de caminos. Lo que se hace es aplanar la estructura del árbol haciendo que cada nodo apunte a la raíz cada vez que se usa Find en él.

Esto es válido, ya que cada elemento visitado en el camino a la raíz es parte del mismo conjunto. El árbol resultante, al ser más plano, acelera las operaciones futuras no sólo en estos elementos, sino también en los que los referencian.

Sin estas heurísticas, la altura de los árboles puede crecer sin control a O(n), lo que implicaría que las operaciones find(x) y merge(x,y) tengan O(n).

El uso de compresión de caminos y union by rank asegura que el tiempo de amortización por operación es sólo $O(m.\alpha(n))$ para m operaciones sobre n elementos, lo cual es óptimo, donde $\alpha(n)$ es la función inversa de Ackermann. Esta función tiene un crecimiento extremadamente lento, por lo que las operaciones de Disjoint-set tienen lugar en un tiempo esencialmente constante.

Aplicaciones:

Los disjoint-sets se suelen utilizar para encontrar los componentes conectados de un grafo no dirigido. Este modelo puede utilizarse entonces para determinar si dos vértices pertenecen al mismo componente, o si la adición de un arco entre ellos daría lugar a un ciclo.

También es un componente clave en la implementación del algoritmo de Kruskal para encontrar el árbol de cubrimiento minimal de un grafo, el cual implementaremos más abajo.