GRAFOS

```
posteriores(X):
    if X está pintado: return ∅
    pintar X
    P ← ∅
    for Y tal que X → Y:
        P ← P ∪ {Y}
        P ← P ∪ posteriores(Y)
    return P

hay ciclo luego de(X):
    if X está pintado de gris: return true
    if X está pintado de negro: return false
    Pintar X de gris
    for Y tal que X → Y:
        if hay ciclo luego de(Y), return true
    Pintar X de negro
    return false

hay ciclo en(G(V, E)):
    for X ∈ V:
        if X está pintado, continue
        if hay ciclo luego de (X):
            return true
    return false
```

P. Dinámica

Conjunto de subprocesos:

- Idealmente polinomial
- Sol. Original a partir de subprocesos
- Orden natural de los subprocesos: peque -> grande
- Recurrencia fácil de calcular

Problema de programación de charlas con ganancias donde queremos maximizar las ganancias totales:

```
rec-opt(j):
    if j = 0:
        return 0
    else:
        if m[j] no está vacía:
            return m[j]
        else:
            m[j] = max{ v_j + rec-opt(b(j)) , rec-opt(j-1) }
            return m[j]
```

`rec-opt`(n) es O(*n*):

```
it-opt:
    m[0] = 0
    for j = 1, 2, ..., n :
        m[j] = max{ v_j + m[b(j)] , m[j-1] }
```

Problema de dar vuelto con monedas de distintos valores donde queremos minimizar la cantidad de monedas a usar:

Modo recursivo:

```
int z(S, n, T):
        si T[S] está en la tabla:
                return T[S]
        si S < 0:
                return infinity
        si S = 0:
                return 0

        minimo = infinity

        for i = 1...n:
                result = z(S-v_i, n, T) +1
                si result < minimo:
                        minimo = result
        T[S] = minimo
        return minimo
```

Modo iterativo:

```
int z(S, n):
        T = [-1 for i = 0...S]
        T[0] = 0
        for i = 1...S:
                si tengo una moneda de costo S:
                        T[i] = 1
                        Break
                minimo = infinity
                for j = 1...n:
                        si v_j ≤ S:
                                result = T[S-v_j] +1
                                si result < minimo:
                                        minimo = result
                T[S] = minimo
```

$$z(S,n) = \begin{cases} \infty \; si \; S < 0 \\ 0 \; si \; S = 0 \\ \min\limits_{i=1..n} (z(S - v_i, n) + 1) \end{cases}$$

DFS

Aristas de árbol: la arista (u, v) es una arista de árbol si v fue descubierto por primera vez al transitar (u, v)

Aristas hacia atrás: aristas (u, v) que conectan un nodo u a un ancestro v en un árbol DFS:

- el grafo es acíclico si y solo si DFS no produce aristas hacia atrás

Aristas hacia adelante: aristas (u, v) que no son de árbol y conectan un nodo u a un descendiente v en un árbol DFS; *no aparecen en grafos no direccionales*

Aristas cruzadas: todas las otras aristas; *no aparecen en grafos no direccionales*

```
dfs(V, E):
    time = 1
    for each u in V:
        u.color = white
    for each u in V:
        if u.color == white:
            time = dfsVisit(u, time)

dfsVisit(u, time):
    u.color = gray
    u.start = time
    time += 1
    for each v in α[u]:
        if v.color == white:
            time = dfsVisit(v, time)
    u.color = black
    u.end = time
    time += 1
    return time
```

Ciclos: comp. fuertemente conectados (CFC)
Acíclicos: Orden topológico (Se obtiene con:)

```
topSort(G)        Si G tiene ciclos, entonces no existe un orden topológico de G

    Crear lista L vacía

    Ejecutar dfs(G) con tiempos

    Insertar nodos en L en orden descendiente de tiempos end

    return L
```

BFS

```
BFS(s): —s es el vértice de partida
    for each u in V-{s}:
        u.color ← white; u.δ ← ∞; π[u] ← null
    s.color ← gray; s.δ ← 0; π[s] ← null
    Q ← cola; Q.enqueue(s)
    while !Q.empty():
        u ← Q.dequeue()
        for each v in α[u]:
            if v.color == white:
                v.color ← gray; v.δ ← u.δ+1
                π[v] ← u; Q.enqueue(v)
        u.color ← black
```

```
Dijkstra(s): —s es el vértice de partida
    for each u in V:
        u.color ← white; d[u] ← ∞; π[u] ← null
    Q ← cola de prioridades
    s.color ← gray; d[s] ← 0; Q.enqueue(s)
    while !Q.empty():
        u ← Q.dequeue()
        for each v in α[u]:
            if v.color == white or v.color == gray:
                if d[v] > d[u] + costo(u,v):
                    d[v] ← d[u] + costo(u,v); π[v] ← u
                if v.color == white:
                    v.color ← gray; Q.enqueue(v)
        u.color ← black
```

O((E+V)LogV)

Heaps

Up

```
sift down(H, i):
    if i tiene hijos:
        i' ← el hijo de i de mayor prioridad
        if H[i'] > H[i]:
            H[i'] ⇄ H[i]
            sift down(H, i')
```

```
extract(H):        —H es el arreglo en que está almacenado el heap
    i ← la última celda no vacía de H
    best ← H[1]
    H[1] ← H[i]
    H[i] ← ∅
    sift down(H, 1)
    return best
```

```
insert(H, e):
    i ← la primera celda desocupada de H
    H[i] ← e
    sift up(H, i)
```

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
QuickSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n^2)$	$O(1)$

	union	*find*
arreglo simple	**O(*n*)**	**O(1)**
lista ligada lineal	**O(1)**	**O(*n*)**
árboles con raíz (representados mediante arreglos)	**O(1)**	**O(*n*)**
árboles con raíz, en que el árbol más pequeño apunta al más grande	**O(1)**	**O(log*n*)**

Codiciosos

```
greedy(a[], n):
    solution = empty
    for i = 1, ..., n:
        x = select(a)
        if feasible(solution, x):
            solution = union(solution, x)
    return solution
```

agrega *x* a l

MST

```
Prim(s): —s es el vértice de partida
    Q ← cola de prioridades;  T ← ∅
    for each u in V-{s}:
        d[u] ← ∞; π[u] ← null; Q.enqueue(s)
    d[s] ← 0; π[s] ← null; Q.enqueue(s)
    while !Q.empty():
        u ← Q.dequeue();  T ← T ∪ (π[u],u)
        for each v in α[u]:
            if v ∈ Q:
                if d[v] > costo(u,v):
                    d[v] ← costo(u,v); π[v] ← u
    return T
```

```
def mst_prim(grafo):
    vertice = grafo.vertice_aleatorio()
    visitados = set()
    visitados.agregar(vertice)
    q = heap_crear()
    arbol = grafo_crear(grafo.obtener_vertices())
    for w in grafo.adyacentes(v):
        q.encolar((v, w) , grafo.peso_arista(v, w))
    while not q.esta_vacia():
        (v, w) = q.desencolar()
        if w in visitados:
            continue
        arbol.agregar_arista(v, w, grafo.peso_arista(v, w))
        visitados.agregar(w)
        for u in grafo.adyacentes(w):
            if u not in visitados: q.encolar((w, u), grafo.peso_arista(w, u))
    return arbol
```

Complejidad prim:
heap binario: O(E log(V))
Matriz abyacencia: O(V^2)

CFC

Las CFCs de un grafo direccional G son conjuntos máximos de nodos $C \subseteq V$ tales que para todo par de nodos u y v en C, u y v son mutuamente alcanzables

Cada CFC tiene un nodo ***representante:***

si el representante de dos nodos es el mismo, entonces los nodos pertenecen a la misma CFC

```
assign(u, rep):
    if u.rep = ∅:
        u.rep = rep
        foreach v in α'[u]:
            assign(v, rep)
```

```
kosaraju(G)

    Crear lista L vacía

    Ejecutar dfs(G) con tiempos

    Insertar vértices en L en orden descendiente de tiempos f

    for each u in L:
        assign(u, u)
```