Backtracking 🌲

La complejidad en algoritmos de backtracking es
$O(\#nodos) \times O(complejidadSubproblema)$, por ejemplo $O(2^k) * O(1) = O(2^k)$

En algunos casos anticipan que ciertos conjuntos de soluciones candidatas del problema no seran factibles y considera evitarlos (podas).

En algunos casos evaluan todas las soluciones candidatas posibles del problema.

Se pueden usar para problemas en los que no se conocen algoritmos polinomiales.

Su complejidad temporal NO es siempre exponencial respecto al tama;o de entrada.

NO explotan el fenomeno de superposicion de problemas.

NO tienen una complejidad temporal estrictamente menor que una solucion por fuerza bruta.

Programacion Dinamica 🤯

Hay superposicion de problemas cuando $O(estadosPosibles) << \Omega(llamRecursivos)$

Ambos enfoques explotan la superposicion de problemas.

Ambas en problemas de optimización combinatoria, además de devolver el valor del óptimo, permiten armar la lista de decisiones que llevan a ese valor.

Ninguna es mas eficiente en complejidad temporal que la otra en todos los casos.

Utilizar memorizacion NO garantiza que solo se calculen los problemas necesarios ni que se reduzca el tiempo de ejecucion (ej. casos de enfoque b-up, donde calculamos demas).

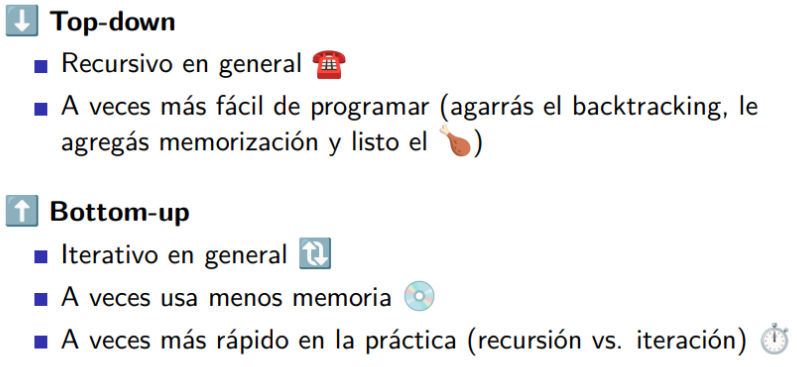
Cuando hay multiples parametros en la funcion implementada NO siempre se necesita una matriz para memorizar los resultados (a veces algunos parametros son constantes o con algunas variables se puede calcular el resto, ej fib(n) = fib(n-1)+fib(n-2)).

Enfoque bottom-up:

- Se construye iterativamente
- Puede ahorrar complejidad espacial
    - Si por ej. solo necesito los ultimos 2 valores f(i-1, x), f(i-2, y) para calcular f(i), armo 2 matrices del tama;o del rango del segundo valor. Por ejemplo si x<=k y y<=k entonces O(k))
- Calcula todos los subproblemas

Enfoque top-down:

- Se construye recursivamente
- Solo calcula los subproblemas que necesita | Su implementación más directa puede, en las circunstancias adecuadas, no requerir computar todas las subinstancias del problema con parámetros más cercanas al caso base,



Divide & Conqueer 🪖⚔️

Teorema maestro:

- Permite resolver relaciones de recurrencia de la forma:
  $T(n) = \begin{cases} \text{a T(n/c)+f(n)} & \text{si} \quad \text{n>1} \\ 1 & \text{si} \quad \text{n=1} \end{cases}$
- Si $f(n) = O(n^{\log_c a - \epsilon})$ para $\epsilon > 0$, entonces $T(n) = \Theta(n^{\log_c a})$
- Si $f(n) = \Theta(n^{\log_c a})$, entonces $T(n) = \Theta(n^{\log_c a} \log n)$
- Si $f(n) = \Omega(n^{\log_c a + \epsilon})$ para $\epsilon > 0$ y $af(n/c) < kf(n)$ para $k < 1$ y n suficientemente grandes, entonces $T(n) = \Theta(f(n))$

Algoritmos Greedy 🍬

Maneras de demostrar greedy:

1) Agarrar una solucion optima y modificarla para que sea greedy manteniendo su optimalidad
2) Agarrar una solucion NO greedy y mostrar que es <= que una solucion greedy. Luego, si la cambio por una greedy sigue siendo optima.
3) Mostrar que cada paso de tu solucion es óptimo y la solucion global (suma de tus pasos) es optima.

Ejemplos:

1. Maximizar streams vistos, sin cambiar de stream antes que termine.

    a. Seleccionar el stream que empiece último (cualquiera si hay más de uno), descartar todos los streams en conflicto con este, y realizar la misma selección para el conjunto de streams restantes.

    b. Seleccionar el stream que termine primero (cualquiera si hay más de uno), descartar todos los streams en conflicto con este, y realizar la misma selección para el conjunto de streams restantes

2. Minimizar costo en estaciones de servicio:

    a. Parar en la estacion de servicio mas cercana a C.

Grafos 🌲

Arbol: grafo conexo sin ciclos.
Puente: Una arista de G tal que G-e tiene mas componentes conexas.
Punto de articulacion: Un vertice de G tal que G-v tiene mas componentes conexas.
Arbol balanceado: Sus hojas estan a nivel H o H-1.
Arbol balanceado completo: Sus hojas estan todas a nivel H.

Arboles:
1. Todo arbol no trivial tiene al menos dos hojas.
2. En un arbol, toda arista es puente.
3. Si G=(V,X) arbol, entonces |X|=|V|-1

Arbol m-ario:
Un arbol m-ario de altura h tiene a lo sumo m^h hojas.
Si es exactamente m-ario tiene m^h hojas.
Tiene $m^{h+1} - 1$ nodos.
Si tiene I hojas, entonces tiene altura h >= $ceil(log_m(I))$
Si T es un arbol exactamente m-ario balanceado no trivial entonces es exactamente h = ceil(logM(I)).

Demostracion inductiva en Grafos:

- **NUNCA:** tomo un grafo G_k de k vértices (o k aristas) y digo que si una propiedad P se prueba para este grafo, agregándole un v vértice (o arista) sigue valiendo P para G_{k+1}. (*)
- **SIEMPRE:** tomo un grafo G_{k+1} de k + 1 vértices (o k + 1 aristas) con ciertas características y le saco un vértice (o arista) con algún tipo de estrategia particular y veo que cumple P. Luego, agrego el vértice o arista y veo que sigue cumpliendo P.

	Matriz de ady.	Lista ady
Construccion	$O(n^2)$	O(n+m)
SonAdyacentes?	O(n)	O(d(v))
Vecinos de v	O(n)	O(d(v))
RemoverUnVerticeYSusAristas	O(m)	O(d(v)*m) ??
sonVecinos(v,w)	O(1)	O(d(v))
listarAristasDelGrafo		O(n+m)
listarDeAristasDeG complemento	O(n^2)	O(n^2)
Devolver una representación de G por lista de adyacencias tal que para todo $v \in V(G)$, N(v) esté ordenado por grado ascendente	O(n+m)	
Util	Grafos densos. Es simetrica entonces se puede guardar solo una mitad de la matriz.	Grafos esparsos.

Factos de Grafos

Un grafo completo K_N tiene $\frac{N*(N-1)}{2}$ aristas.

Facto: Toda orientacion aciclica de un grafo tiene al menos un sumidero.

Facto: Todo grafo etiquetado con al menos una arista tiene una cantidad par de orientaciones aciclicas distintas.

Facto: Toda orientacion aciclica de Kn tiene un unico sumidero.

Facto: Si a una orientacion acıclica H de un grafo le agregamos un nuevo vertice v junto a una arista u → v para todo vertice u de H, entonces el grafo resultante es la orientacion aciıclica de un grafo.

Facto: Un grafo Kn tiene n! orientaciones aciclicas.

Facto: Hay un numero par de orientaciones aciclicas para un grafo (si las hay).

Definicion:
$G = (V, E) \rightarrow G^c \equiv^{def} (V, E')$. Donde E' tiene todas las aristas que G no tiene.

$K_n = (V, E \cup E'), |V| = n$

Facto: G no conexo → G^c conexo

NO vale que si G es un grafo conexo entonces no es un grafo junta.

NO vale que la existencia de un ciclo en un grafo asegura que el grafo es conexo.

NO vale que todo grafo etiquetado tiene al menos un sumidero.

Facto: El digrafo resultado de darle una orientacion a un grafo completo tiene un camino de longitud n − 1.

- *Demo: Sale con induccion de cantidad de vertices. En el paso inductivo D' tiene n-1 vertices y es la orientacion aciclica de Kn+1, si le sacamos un vertice v, por HI sabemos que existe una camino P de long n-1. Cuando le agregamos v, sabemos que v se conecta con todos los vertices, si todas las aristas fuesen hacia v, agregamos v al final del camino de P. Sino, agarramos al primer vertice u de P en el cual v→u esta en D', luego sabemos que el anterior w tiene a w→v y extendemos a P como* $P' = p_0 \rightarrow w \rightarrow v \rightarrow p_{n-1}$.
  Como $P = p_0 \rightarrow w \rightarrow .. \rightarrow p_{n-1}$ *con |P|=n-1 entonces |P'|=n.*

Facto: Si G tiene exactamente dos vertices de grado impar hay camino entre ellos.

- Verdadero. Lo vemos por el contrarreciproco,

  d(w) y d(v) impares y no existe camino entre v y w → G no tiene exactamente 2 vertices de grado impar.

  De la premisa vale que v y w estan en diferente componente conexa y tienen grado impar. Pero.. puede haber un unico nodo de grado impar en una componente conexa? No.

  Porque sabemos que $\sum_{v \in V} d(v) = 2|E|$ y $d(v_o) \equiv 1(mod\ 2)$, entonces

  $\sum_{v \in V} d(v) = d(v_0) + \sum_{v \in V-\{v_0\}} d(v) = 2|E|$, luego si miramos las congruencias en modulo

  dos vemos: $2|E| \equiv 0\ (mod\ 2)$ y por consecuencia $d(v_0) + \sum_{v \in V-\{v_0\}} d(v) \equiv 0$ con

  $d(v_0) \equiv 1(mod\ 2)$, luego$1 + \sum_{v \in V-\{v_0\}} d(v) \equiv 0\ (mod\ 2) \Leftrightarrow \sum_{v \in V-\{v_0\}} d(v) \equiv 1(mod\ 2)$

  Pero por hipotesis, d(x) es par para todo x en V\{v0} (porque v0 es el unico nodo de la componente con grado impar), luego esto es imposible. Entonces, **no puede haber un unico nodo de grado impar en una componente conexa**.

Facto: No puede haber un unico nodo de grado impar en una componente conexa.

Teorema: Un grafo es bipartito ⇔ no tiene ciclos de long. impar.

Corolario: Si G=(V,X) no tiene ciclos (circ. simples) y tiene c componentes conexas, entonces |X|=|V|-c

Corolario: Si G=(V,X) no tiene ciclos (circ. simples), entonces |X|=|V|-1

Lema: Si G=(V,X) es un grafo conexo y e es una arista de X,
entonces G-e es conexo ⇔ e esta en un ciclo de G

Teorema: Todo grafo conexo tiene al menos un AG.

Teorema: Si G es conexo y tiene un unico AG → G es arbol.

Teorema: Si T=(V,X1) es un AG de G=(V,X0) y E es una arista de X0 - X1, entonces T'=T+e-f es un AG de G, con G una arista del unico circuito de T+e.

Teorema: $\sum_{v \in V(G)} d(v) = 2m$

Lema: Una arista (u, v) de un grafo G es puente si y solo si no hay camino entre u y v en G \ (u, v).

Lema: Sea G un grafo conexo. Toda arista de G en un recorrido DFS es o una tree edge o una backward edge.

Lema: Sea G conexo, E es un puente de G ⇔ E no esta en un ciclo de G.

Util: Sea G tal que |V(G)|=2(N+1) y {v,w} es una arista en E(G) → G'=(G-v)-w tiene |V(G')|=2N y |E(G')|=|E(G)|-d(v)-d(w)+1

Util:

$M \geq \frac{(N-2)*(N-1)}{2}$ → G conexo

$M <= \frac{N*(N-1)}{2}$

Lema: Arista es puente ⇔ no pertenece a ningun ciclo

Lema: Una arista (u, v) de un grafo G es puente si y solo si no hay camino entre u y v en G - (u, v).

- *La idea de la demo (la original es larguisima) es que:*
  <=) Si no hay camino entre v y w en G-{uv} entonces u y v estan en distintas componentes conexas, por lo que G tiene menos componentes conexas que G-{uv}, entonces uv es puente.
  =>) Si uv puente, entonces G-{uv} tiene mas componentes conexas que G. En particular, esto sucede si solo si los vertices que se unen estaban en distintas componentes conexas. Por lo tanto, no habia camino entre ellos en G-{uv}

Teorema de Robbins: Este teorema establece que un grafo no dirigido tiene una orientación fuertemente conexa si y solo si es biconexo.

Biconexo: Un grafo es biconexo si no contiene vértices de corte, es decir, su eliminación (junto con las aristas incidentes) no aumenta el número de componentes conexas del grafo.

Propiedad: Dado G = (V, E) un grafo conexo y Tr = (V, ET) un arbol DFS de G enraizado en r ∈ V, r es punto de articulacion de G si y solamente si r tiene grado mayor a 1 en T

Propiedad: Un nodo interno v del arbol Tr es punto de articulacion si y solamente si alguno de los subarboles enraizados en sus hijos no tiene ejes hacia nodos que son ancestros de v.

Algoritmos BDS/DFS

Complejidad optima con lista de adyacencias.

Conceptos:

Un **recorrido** de un grafo es una secuencia alternada de vertices y aristas. En los grafos (no multi ni pseudo) basta con aclarar los vertices.

Un **camino** es un recorrido que no pasa dos veces por el mismo vertice.

Una **seccion** es una subsecuencia de un camino.

Un **circuito** es un recorrido que empieza y termina en el mismo vertice.

Un **circuito simple o ciclo** es un circuito de 3 o mas vertices que no pasa dos veces por el mismo vertice.

Teorema (de la teorica):
Sea G un grafo conexo. Toda arista de G en un recorrido DFS es o una tree edge o una backward edge.

Cuando se realiza un recorrido DFS sobre un grafo conexo no dirigido G, se obtiene un árbol T que cumple que toda arista (v, w) de G que no está en T es una back-edge, es decir, que v es ancestro de w o viceversa. O sea…

Lema: Sea G conexo y T un AG por DFS. E=(v,w) perteneciente a E(G)-E(T) implica que v es ancestro de w en T (o viceversa).

En digrafos…
Si hacemos el algoritmo de DFS sobre un grafo dirigido D, desde un vértice r que alcanza a todos los vértices, obtenemos un árbol T′ . Una arista dirigida v → w de D que no pertenece a T' puede ser:

- BE, FE o CrossEdge a izquierda.

Lema:
Una tree edge de un arbol DFS T de un grafo conexo G es un puente si y solo si no hay ninguna backward edge que la cubra

Lema:
Una backward edge no puede ser un puente.

Calcular la distancia entre cualquier par de nodos linealmente:
La idea es hacer BFS sobre G y una raiz arbitraria v. Durante el recorrido te guardas la profundidad de cada vertice y el padre de cada uno de ellos. Luego, para obtener la distancia entre cualquier par de vertices x,y hay dos casos:

1. $x, y \in V(G) - \{v\}$: Aca nos fijamos en O(n) el primer ancestro comun K entre x e y, contando la longitud de los caminos de d(x,K) y d(y,K).
   Vale que $d(x, y) = d(x, K) + d(K, y)$.
2. x o y son v (cuesta O(1) en cualquier subcaso):
   a. x=y=v → d(x,y)=0
   b. x=v, $y \in V(G) - \{v\}$ → d(x,y)=depth(y).

Esto cuesta $\theta(n + m)$.

Determinar si un grafo tiene ciclos:
Aca tenemos que fijarnos si el grafo tiene BE. Esto lo hacemos clasificando cada arista a partir del diagrama de tiempos con el recorrido DFS.
Si alguna arista es BE → el grafo tiene al menos un ciclo.
Esto cuesta $\theta(n + m)$.

Hay camino entre v y w?
Si nos interesa saber si hay camino entre v,w en un Grafo G' = G\puntosDeArticulacion(G). Tenemos que ver si v,w estan en la misma componente conexa. Esto lo hacemos con DFS.

Encontrar puntos de corte en G:

c) Dado un árbol DFS T_r de G (que se puede encontrar en tiempo lineal representando el grafo G como lista de adyacencias) es sencillo decidir si su raíz es punto de articulación en tiempo $O(1)$ usando el inciso a), asi que solo nos falta ver como "implementar" el inciso b) en tiempo lineal.

Escribamos como $depth(v)$ a la profundidad del nodo v en el árbol T_r (es decir, $depth(r) = 0$, y los hijos de r tienen profundidad 1). Para resolver este problema nos gustaría saber, para cada subárbol, la menor profundidad a la que puede acceder tomando alguna *back edge*. Para esto, notemos como $low(w)$ a la menor profundidad que se puede alcanzar empleando alguna *back edge* del subárbol enraizado en w.

Supongamos que logramos calcular estos valores $depth(\cdot)$ y $low(\cdot)$ para todos los nodos, y que tenemos estos valores guardados en un vector. Luego, podemos decidir si un nodo cualquiera v es punto de articulación comparando $depth(v)$ con $low(w_i)$ para cada uno de sus hijos $w_1, \ldots, w_k$. Más precisamente, v será punto de articulación si para alguno de sus hijos w_i vale que $depth(v) \geq low(w_i)$ (i.e. el subárbol enraizado en w_i no tiene ninguna *back edge* hacia nodos ancestros de v). Para checkear esto tenemos que hacer, por cada nodo v, a lo sumo $O(d(v))$ operaciones (usando como representación lista de adyacencias). Por lo tanto, podemos checkear todos los nodos en $O(\sum_{v \in V} d(v)) = O(m)$, que es lineal en el tamaño de entrada.

Entonces solo nos falta explicar cómo calcular estos vectores con los valores $depth(\cdot)$ y $low(\cdot)$ dado el árbol T_r. Los valores de $depth(\cdot)$ ya los calcula de por sí el DFS, por lo que solo falta ver cómo calcular $low(\cdot)$ en tiempo lineal. Observemos que vale

$$low(v) = \text{mín}(\min_{vz \in BE(v,T_r)} depth(z), \min_{w \in H(v,T_r)} low(w)) \quad (1)$$

donde $H(v, T_r)$ es el conjunto de hijos de v en el árbol T_r, y $BE(v, T_r)$ es el conjunto de *back edges* de T_r que inciden en v[2]. Es decir, la menor profundidad que se puede alcanzar usando una *back edge* del árbol enraizado en v se alcanza usando alguna *back edge* que incide en v (lado izquierdo del mínimo), o bien usando alguna que incide en algún nodo en uno de los subárboles de sus hijos (lado derecho del mínimo).

[2]Y, técnicamente, tales que $depth(z) < depth(v)$

Esta relación recursiva ya nos da un algoritmo para calcular los valores de $low(\cdot)$. Para hacerlo de forma eficiente podemos usar programación dinámica sobre un vector de n posiciones: en ese caso, el costo de calcular todos los valores se puede computar como la sumatoria del costo de calcular cada valor, asumiendo que los llamados recursivos se resuelven en tiempo constante. Dado un nodo v cualquiera calcular $low(v)$ usando la ecuación 1 requiere recorrer todos los ejes que inciden en él, lo cual toma $O(d(v))$ (de nuevo, usando como representación lista de adyacencias). Por lo tanto, la complejidad final para resolver la recursión es $O(\sum_{v \in V} d(v)) = O(m)$. También deberíamos agregar un $O(n)$ para iniciar el vector donde se guardan los valores de $low(\cdot)$.

Resumiendo, nuestro algoritmo completo es el siguiente:

1. Dado G como lista de aristas, construimos una nueva representación de G como lista de adyacencias[3]. $O(n+m)$
2. Armamos un árbol DFS T de G, obteniendo en particular los valores de $depth(\cdot)$. $O(n+m)$
3. Calculamos con programación dinámica los valores $low(\cdot)$ sobre T. $O(n+m)$
4. Recorremos los nodos y para cada uno decidimos si es o no punto de articulación, usando el inciso a) o bien el b), de acuerdo a si es la raíz o no. $O(n+m)$

En base a los incisos a) y b) sabemos que este algoritmo es correcto, y aparte tiene la complejidad pedida.

IMPORTANTE (para alguno):

cada uno de sus hijos $w_1, \ldots, w_k$. Más precisamente, v será punto de articulación si para alguno de sus hijos w_i vale que $depth(v) \geq low(w_i)$ (i.e. el subárbol enraizado

O sea, si alguno de sus hijos no tiene BE → es punto art.

Encontrar una componente fuertemente conexa de G:

Verificar si G es biconexo (es decir, si no tiene puntos de corte).
Si G es biconexo, orientar las aristas con la direccion de un recorrido DFS nos devuelve un D(T) fuertemente conexo.
Si no es biconexo, no hay un D(T) fuertemente conexo.

Identificar aristas con DFS + tiempos de visita.

Tree-edges: aristas visitadas por primera vez.
Back-edges: arista u,v con v ANCESTRO de u y post[v] < post[u].
Cross-edges: las demás.
Forward?

Encontrar puentes en G:

Una tree edge de un arbol DFS T de un grafo conexo G es un puente si y solo si no hay ninguna backward edge que la cubra.

Perfecto, entonces la idea del algoritmo sería esta:

1. Hacer un recorrido DFS sobre el grafo G desde un vértice cualquiera.
2. Identificar las tree edges y backward edges de G.
3. Para cada tree edge (u, v), ver si hay alguna backward edge que la cubra. Si no hay ninguna, entonces (u, v) es un puente. Sino, no lo es.

Para el 3 usamos programacion dinamica:

$$\text{minNivelCubierto}(v) = \text{mín} \left\{ \begin{array}{l} \text{nivel}(v), \\ \text{minBackwardEdge}(v), \\ \text{mín}_{\texttt{hijo} \in \text{hijos}(v)} \{\text{minNivelCubierto}(\texttt{hijo})\} \end{array} \right\}.$$

si $\text{minNivelCubierto}(v) < \text{nivel}(v)$, la arista (u, v) está cubierta, sino, no.

Importante: Sea (u,v) si hay alguna BE que la cubre (low(v) < depth(v))→ no es puente.

BFS	DFS
Encontrar componentes fuertemente conexas	Encontrar componentes fuertemente conexas
Ver si un grafo es conexo	Ver si un grafo es conexo
Encontrar distancias aristas uno a todos	Encontrar puentes/puntos de articulacion
Ver si un digrafo tiene ciclos	Ver si un digrafo tiene ciclos

Orden topologico

Lema 1: Un DAG tiene al menos un sumidero.

Lo obtenemos con el algoritmo de Kahn en O(n+m)

Tambien lo podemos obtener de la siguiente forma:

1. Obtenemos los grados de salida de cada nodo en O(n+m)
2. Armamos una cola con los nodos de grado de salida 0 en O(n)
3. Sacamos un nodo de la cola y le restamos uno al grado de salida de sus padres.
4. Validamos si hay que agregar a los padres a la cola.
5. Repetimos hasta que la cola esta vacia.