Brute Force

- Un algoritmo de fuerza bruta para un problema de optimización combinatoria consiste en generar todas las soluciones factibles y quedarse con la mejor.
- Habitualmente, un algoritmo de fuerza bruta tiene una complejidad exponencial.

Backtracking

- Recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema computacional.
- Las nuevas soluciones parciales son sucesoras de la anterior.
- En cada paso se extienden las soluciones parciales.
- Se puede pensar este espacio como un árbol dirigido..
- Permite descartar configuraciones antes de explorarlas (podar el árbol).
- Se pueden usar en problemas para los que no se conoce un algoritmo polinomial.
- En algunos casos anticipan que ciertos subconjuntos de soluciones candidatas del problema no serán factibles y evita considerarlos.
- En algunos casos evalúan todas las soluciones candidatas posibles del problema.

Dynamic Programming

- Se divide el problema en subproblemas más pequeños.
- Superposición de estados: El árbol de llamadas recursivas resuelve el mismo problema varias veces, es decir que se realizan muchas veces llamadas a la función recursiva con los mismos parámetros. Esto sucede cuando la cantidad de subproblemas son menores a la cantidad de llamadas recursivas.
- Un algoritmo de programación dinámica evíta estas repeticiones.
- Explota el fenómeno de superposición de subproblemas.

Top-Down

- Se implementa recursivamente, pero se guarda el resultado de cada llamada recursiva en una estructura de datos (memoización). Si una llamada recursiva se repite, se toma el resultado de esta estructura.
- Su implementación más directa puede, en las circunstancias adecuadas, no requerir computar todas las subinstancias del problema con parámetros más cercanas al caso base.
- En problemas de optimización combinatoria, además de devolver el valor del óptimo, permite armar la lista de decisiones que llevan a ese valor.

Bottom-Up

- Resolvemos primero los subproblemas más pequeños y guardamos (habitualmente en una tabla) todos los resultados.
- Permite, en las circunstancias adecuadas, ahorrar complejidad espacial.
- En problemas de optimización combinatoria, además de devolver el valor del óptimo, permite armar la lista de decisiones que llevan a ese valor.

Divide & Conquer

- Dividir un problema en subproblemas más pequeños del mismo tipo que el original.
- Resolver los problemas más pequeños.
- Combinar las soluciones

Greedy

- Construir una solución seleccionando en cada paso la mejor alternativa, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.
- Para algunos problemas, un algoritmo de backtracking puede encontrar una solución mejor que la provista por la estrategia greedy.
- Todos los problema se pueden resolver mediante Greedy pero no asegura que sea la más óptima
- Habitualmente, proporcionan heurísticas sencillas para problemas de optimización.
- En general permiten construir soluciones razonables (pero subóptimas) en tiempos eficientes.
- Una **heurística** es un procedimiento computacional que intenta obtener soluciones de buena calidad para un problema, intentando que su comportamiento sea lo más preciso posible.

Teorema Maestro

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

n: es el tamaño del problema.
a: es el número de subproblemas en la recursión.
n/b: es el tamaño de cada subproblema. (Todos los subproblemas tienen el mismo tamaño.)
f(n): es el costo del trabajo realizado fuera de las llamadas recursivas, que incluye el costo de la división del problema y el costo de la unión de las soluciones de los subproblemas.

1. Si $f(n)$ es $O(n^c)$, donde $c < \log_b(a)$, entonces el tiempo de ejecución es $\Theta(n^{\log_b(a)})$.
2. Si $f(n)$ es $\Theta(n^{\log_b(a)})$, entonces el tiempo de ejecución es $\Theta(n^{\log_b(a)} \log n)$.
3. Si $f(n)$ es $\Omega(n^c)$, donde $c > \log_b(a)$, si $a\, f(n/b) \leq k\, f(n)$ para alguna constante $k < 1$ y suficientemente grandes n, entonces el tiempo de ejecución es $\Theta(f(n))$.

Si existe un entero k tal que $g(n)$ es $O(n^k)$ se puede demostrar que:

$$t(n) \text{ es } \begin{cases} O(n^k) & \text{si } r < b^k \\ O(n^k \log n) & \text{si } r = b^k \\ O(n^{\log_b r}) & \text{si } r > b^k \end{cases}$$

Grafos

G es un **grafo** representado por un par **(V,E)**

F es un **digrafo** representado por un par **(V,E)** donde (v,w) ≠ (w,v) donde v,w ∈ **V(G**)

V(G) = Conjunto de Vértices

E(G) = Conjunto de Aristas

Recorrido: una sucesión de vértices y aristas del grafo

Camino: un recorrido que no pasa dos veces por el mismo vértice.

Circuito: un recorrido que empieza y termina en el mismo nodo

Ciclo o circuito simple: un circuito que no repite vértices. (Nota: para grafos, no consideramos como válido al ciclo de longitud 2)

Longitud: la longitud de un recorrido se nota l(P) y es la cantidad de aristas del mismo.

Distancia entre dos vértices: longitud del camino más corto entre los vértices (si no existe se dice ∞).

Adyacencia: Un nodo es adyacente a otro si están conectados.

Indidencia: Una arista es incidente a un nodo si conecta dicho nodo con algún otro.

n: Cantidad de vértices

m: Cantidad de aristas

Para grafos:

- **N(v):** vecindario del vértice v (conjunto de nodos adyacentes a v).
- **N[v]:** vecindario cerrado de v, N[v] = N(v) ∪ v .
- **d(v)**: el grado de v (cantidad de vecinos), d(v) = |N(v)|.

Para digrafos:

- **Nin(v)** y **Nout (v)** son los vecindarios de entrada y salida respectivamente.
- **din(v)** y **dout (v)** son los grados de entrada y salida respectivamente.

Podríamos **representar** el grafo de dos maneras:

- **Conjunto de aristas:** guardamos el conjunto **E(G)** del grafo.
- **Diccionario:** le asociamos a cada vértice “v” su vecindario **N(v)**.

Estructura de representación:

- **Lista de aristas**
- **Lista de adyacencia**
- **Matriz de adyacencia**

Complejidades

Las complejidades para las representaciones vistas quedarían:

	lista de aristas	matriz de ady.	listas de ady.
construcción	$O(m)$	$O(n^2)$	$O(n+m)$
adyacentes	$O(m)$	$O(1)$	$O(d(v))$
vecinos	$O(m)$	$O(n)$	$O(d(v))$
agregarArista	$O(m)$	$O(1)$	$O(d(u)+d(v))$
removerArista	$O(m)$	$O(1)$	$O(d(u)+d(v))$
agregarVértice	$O(1)$	$O(n^2)$	$O(n)$
removerVértice	$O(m)$	$O(n^2)$	$O(n+m)$

BFS

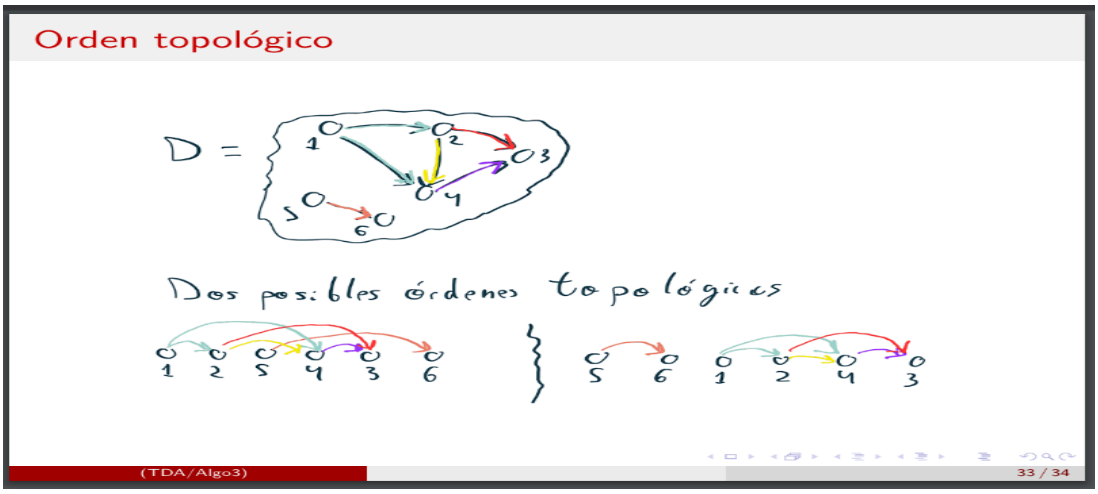
- Nos devuelve un árbol v-geodésico
- Encuentra la solución más corta en términos de número de pasos de un nodo al nodo enraizado, si existe.
- Es útil para encontrar el camino más corto en gráficos no ponderados.
- Puede determinar si un grafo es bipartito.
- Complejidad: O(m+n)

DFS

- Nos devuelve el árbol o bosque (como un vector de padres)
- Mejor para detectar ciclos
- Detecta backedges
- Complejidad: O(m+n)

Orden Topológico

Dado un digrafo D, un orden topológico de D es un ordenamiento v1 . . . vn de sus nodos que cumple que todo eje queda de la forma vi vj con i < j (en el ordenamiento). Es decir, damos un orden a los nodos de tal forma que las aristas apuntan de izquierda a derecha ("no hay aristas para atrás").



Algoritmo

- Primero tenemos que verificar si el digrafo tiene ciclos (Ejercicio, sale con DFS). Si tiene ciclos no tiene orden topológico.
- Usamos la implementación de DFS de tres estados: no lo vi, empecé a ver y termine de ver.
- Vamos a modificar DFS para que si termina de procesar un nodo lo pusheen a un stack finish.

Grafo Conexo

- Un grafo se dice conexo si existe camino entre todo par de vértices.
- Una componente conexa de un grafo G es un subgrafo conexo máximo de G .

Grafo Bipartito

- Un grafo G = (V , X) se dice bipartito si existen dos subconjuntos V1, V2 del conjunto de v´ertices V tal que:

$$V = V1 \cup V2, \qquad V1 \cap V2 = \varnothing$$

  y tal que todas las aristas de G tienen un extremo en V1 y otro en V2.

- Un grafo bipartito con subconjuntos V1, V2, es bipartito completo si todo vértice en V1 es adyacente a todo v´ertice en V2
- Un grafo G es bipartito si, y sólo si, no tiene ciclos de longitud impar.

Punto de articulación:

Un punto de articulación en un grafo no dirigido es un vértice cuya eliminación aumenta el número de componentes conectados en el grafo.

Tiene que tener por lo menos grado 2

Puente:

Un puente es una arista en un grafo que, si se elimina, aumenta el número de componentes conectados en el grafo.

Un puente no pertenece a ningún ciclo.

Clasificación de aristas

Grafos No Dirigidos

Backedge:

Es una arista que no está en el árbol pero si está en el grafo, con el cual se puede ir hacia arriba o hacia abajo en el grafo conectando vértices que no están conectados en el árbol.

Grafos Dirigidos

Tree-edge:

Son aristas del bosque $G\pi$ en profundidad. La arista (u, v) es una arista de árbol si v se descubrió primero explorando la arista (u, v).

Back-edge:

Son las aristas (u, v) que conectan un vértice u con un antepasado v en un bosque de primero en profundidad. v en un árbol de profundidad. Consideramos que los bucles propios, que en los grafos dirigidos.

Forward-edge:

Son las aristas que no pertenecen a un árbol (u, v) y que conectan un vértice u a un descendiente propio v en un árbol de profundidad-primera.

Cross-edge:

Son todas las demás aristas. Pueden ir entre vértices de vértices del mismo árbol de profundidad, siempre que un vértice no sea del otro, o pueden ir entre vértices de diferentes árboles de profundidad. diferentes.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]      // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6      return A[q]      // the pivot value is the answer
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

Theorem 9.2
The procedure RANDOMIZED-SELECT on an input array of n distinct elements has an expected running time of $\Theta(n)$.

```
SELECT(A, p, r, i)
1   while (r − p + 1) mod 5 ≠ 0
2       for j = p + 1 to r                  // put the minimum into A[p]
3           if A[p] > A[j]
4               exchange A[p] with A[j]
5       // If we want the minimum of A[p : r], we're done.
6       if i == 1
7           return A[p]
8       // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
9       p = p + 1
10      i = i − 1
11  g = (r − p + 1)/5                       // number of 5-element groups
12  for j = p to p + g − 1                  // sort each group
13      sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14  // All group medians now lie in the middle fifth of A[p : r].
15  // Find the pivot x recursively as the median of the group medians.
16  x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17  q = PARTITION-AROUND(A, p, r, x)   // partition around the pivot
18  // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19  k = q − p + 1
20  if i == k
21      return A[q]                         // the pivot value is the answer
22  elseif i < k
23      return SELECT(A, p, q − 1, i)
24  else return SELECT(A, q + 1, r, i − k)
```

Theorem 9.3
The running time of SELECT on an input of n elements is $\Theta(n)$.

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max {q, p[i] + CUT-ROD(p, n − i)}
return q
```

$T(n) = 2^n$,

```
MEMOIZED-CUT-ROD(p, n)
let r[0 : n] be a new array          // will remember solution values in r
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0                          // already have a solution for length n?
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n                   // i is the position of the first cut
        q = max {q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r)}
r[n] = q                             // remember the solution value for length n
return q

BOTTOM-UP-CUT-ROD(p, n)
let r[0 : n] be a new array          // will remember solution values in r
r[0] = 0
for j = 1 to n                       // for increasing rod length j
    q = −∞
    for i = 1 to j                   // i is the position of the first cut
        q = max {q, p[i] + r[j − i]}
    r[j] = q                         // remember the solution value for length j
return r[n]
```

The running time of BOTTOM-UP-CUT-ROD is $\Theta(n^2)$.