SEMANA 1: ÁRBOLES DE BÚSQUEDA AVANZADOS

ÁRBOLES BINARIOS DE BÚSQUEDA

- La raíz es mayor que sus hijos del lado izquierdo y menor que sus hijos del lado derecho.
- Recursivamente, los dos hijos son árboles binarios de búsqueda.
- En el caso de eliminar un nodo interno, se sustituye por el nodo anterior o posterior del inorden (Izquierda-Raíz-Derecha).

Operaciones:

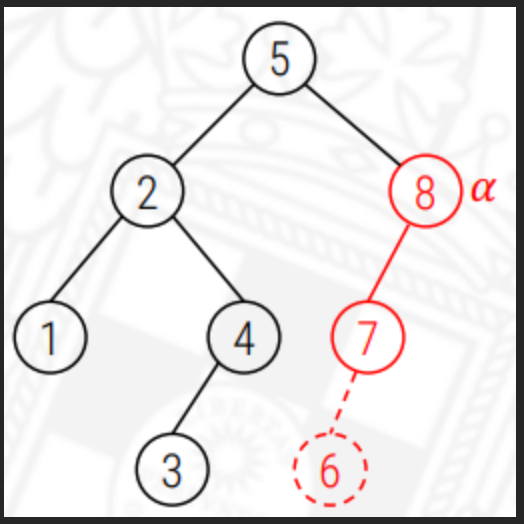
- Búsqueda: O(n), siendo n la altura.
- Inserción: O(n), siendo n la altura.
- Eliminación: O(n), siendo n la altura.
    - Caso medio: O(log N), siendo n el número de nodos.

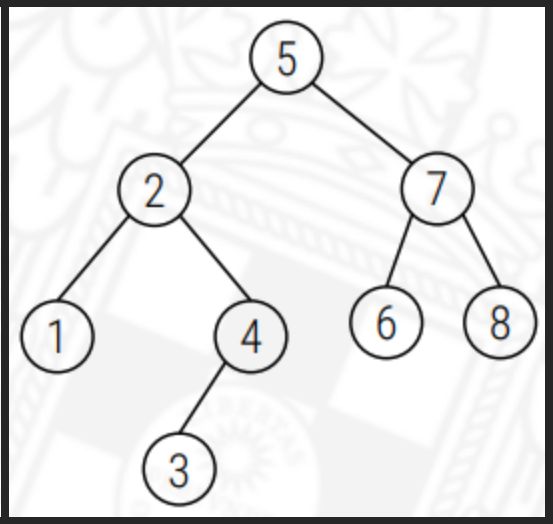
ÁRBOLES EQUILIBRADOS (AVL)

- Todos los nodos cumplen que la diferencia de altura de sus hijos es como mucho 1.
- La altura está en **O(log N)**, siendo n el número de nodos.
- El mínmo número de nodos en un árbol de altura h: **S(h) = S(h - 1) + S(h - 2) + 1, con S(0) = 0 y S(1) = 1**.

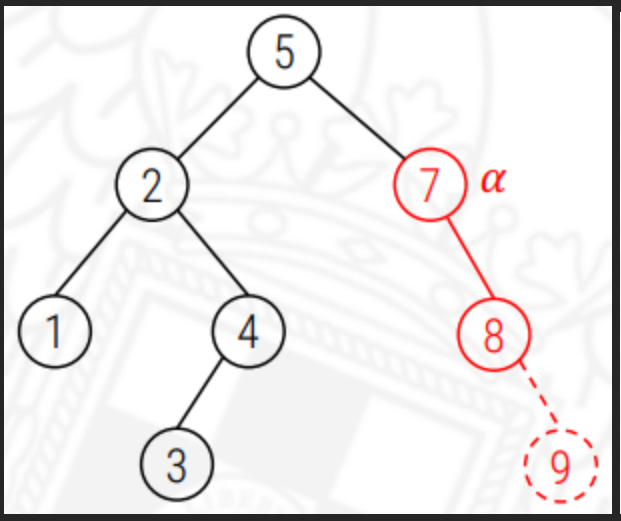
Operaciones:

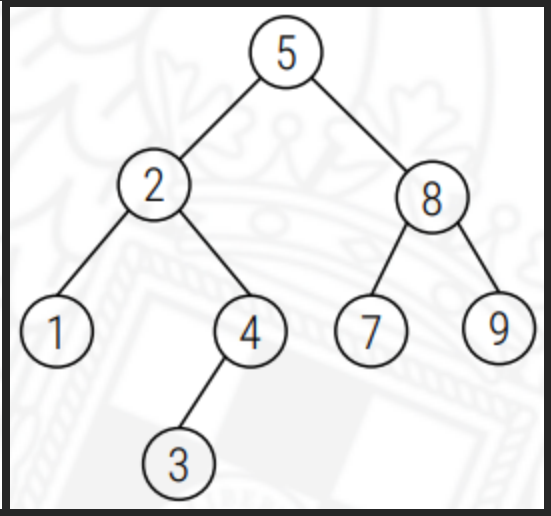
- Si insertamos a la izquierda y denuevo a la izquierda --> Rotación simple a la derecha:



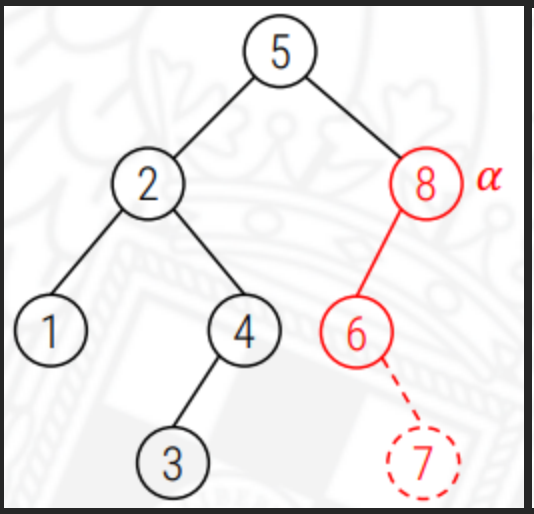


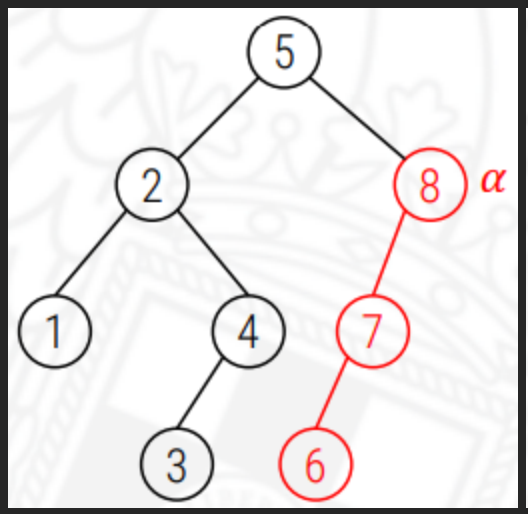
- Si insertamos a la derecha y denuevo a la derecha --> Rotación simple a la izquierda:

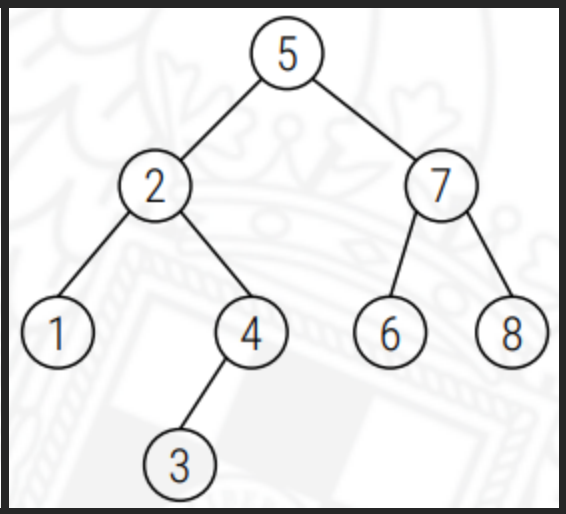




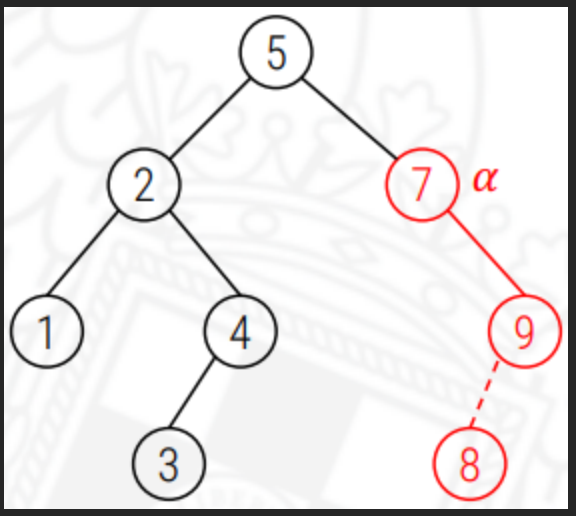
- Si insertamos a la derecha y luego a la izquierda --> Rotación doble izquierda-derecha:

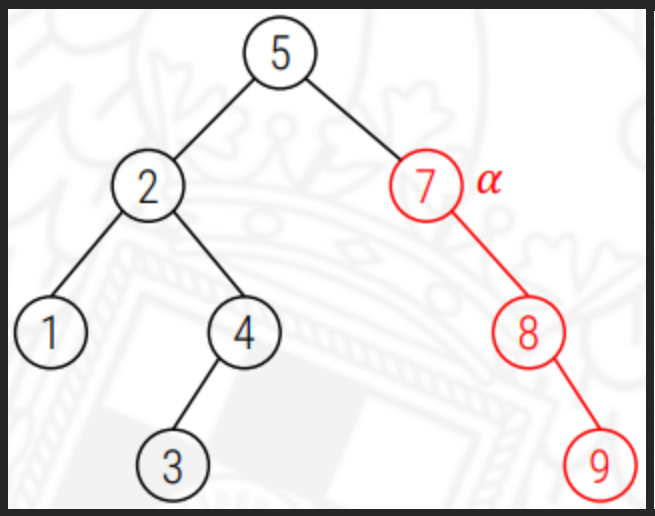


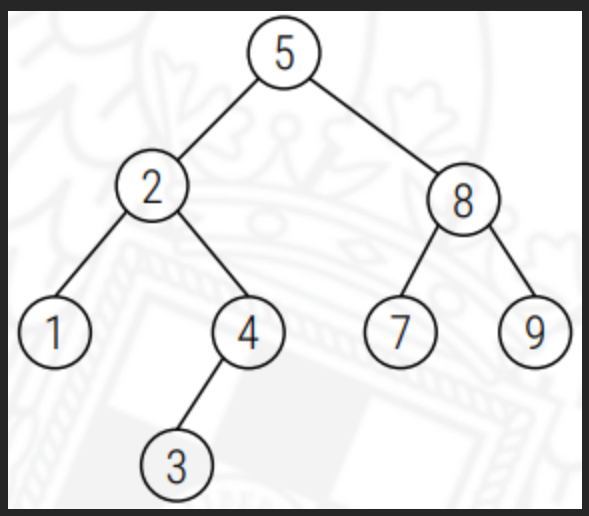




- Si insertamos a la izquierda y luego a la derecha --> Rotación doble derecha-izquierda:







TESTS

- Cuando se elimina un nodo (árbol jodido), hacer una **rotación contraintuitiva** con el hijo más grande y luego arreglarlo con una rotación hacia el lado opuesto.
- Cuando se inserta un nodo, varios de los nodos en la rama desde la raíz pueden perder la condición de equilibrio.
  - Si se realiza **una rotación**, se recupera la altura anterior a la inserción.
- Cuando se elminina un nodo, varios de los nodos en la rama desde la raíz pueden perder la condición de equilibrio.
  - Pueden hacer falta **varias rotaciones** para equilibrar el árbol.
- Para insertar una secuencia ordenada crecientemente de valores y no realizar rotaciones, se debe usar una estrategia recursiva: primero la mediana de la secuencia y luego el mismo proceso para las dos mitades.
- Para **eliminar el menor elemento de un árbol**, hace falta bajar por la izquierda lo máximo posible, lo que en el caso peor será la altura del árbol --> **Coste O(log N)**.
- El coste de un **algoritmo que ordena** N valores insertándolos primero en un **AVL** y recorriéndolo posteriormente es de **O(N log N)**. Cada inserción está acotada superiormente por O(log N), por lo que N inserciones tendrán un coste de O(N log N). Recorrer el árbol en inorden tiene un coste de O(N).
- El coste de un **algoritmo que ordena** N valores insertándolos primero en un **ABB** y recorriéndolo posteriormente es de **$O(N^2)$**. Cada inserción puede tener un coste O(N), por lo que N inserciones tendrán un coste de $O(N^2)$. Recorrer el árbol en inorden tiene un coste de O(N).
- El número mínimo de nodos de un árbol equilibrado de altura h es Fib(h + 2) - 1.
- La mayor diferencia entre los hijos de un árbol AVL puede ser $2^{h - 1}$ - 1 - Fib(h) - 1.

SEMANA 2: COLAS DE PRIORIDAD Y MONTÍCULOS

COLAS DE PRIORIDAD

- Cada elemento tiene asignada una prioridad, que determina en qué orden va a ser atendido.
- Según el orden, pueden ser colas de prioridad de mínimos o máximos.

Implementaciones

implementación	push	top	pop
vector desordenado	1	N	N
vector ordenado	N	1	1
montículo binario	$\log N$	1	$\log N$
montículo k-ario	$\log_k N$	1	$k \log_k N$
Fibonacci	1	1	$\log N$
imposible	1	1	1

MONTÍCULOS BINARIOS

- Los montículos son árboles semicompletos equilibrados en altura.
- Un árbol binario es semicompleto cuando tiene una serie de posiciones vacantes en el último nivel y agrupadas a la derecha.

Propiedades

- Completos:
  - Tienen 2^{i-1} nodos, siendo i el nivel.
  - Tienen 2^{h-1} hojas, siendo h la altura.
  - Tienen 2^{h-1} nodos, siendo h la altura.
- Semicompletos:

- Su altura es [log n] + 1, siendo n el número de nodos.
- Dado un padre i, sus **hijos se encontrarán en las posiciones 2i y 2i + 1**.
- Dado un hijo i, su **padre se encontrará en la posición i/2** (división entera).

Inserción

Se ocupa la primera hoja disponible y se comprueba si está bien colocado el elemento a insertar. En caso de que no lo esté, se flota la posición hasta que esté ordenado. Coste: O(log N), siendo n el número de nodos.

Eliminación

Se libra la última hoja ocupada y se comprueba si el elemento que la ocupaba está bien colocado en la raíz del árbol. En caso de que no lo esté, se hunde la posición hasta que este ordenado, sustituyendolo siempre por el más prioritario.

TESTS

- Un arbol binario semicompleto de k niveles tiene como mínimo 2^{k-1} nodos.
- El coste, en el caso peor, de **eliminar el elemento más prioritario** es de **O(log N)**, pues éste es sustituido por el último elemento del último nivel, y puede hacer falta hundirlo tantos niveles como tiene el árbol.
- El coste, en el caso peor, de **insertar un nuevo elemento** es de **O(log N)**, pues es introducido en la última posición del último nivel y puede hacer falta flotarlo tantos niveles como tiene el árbol.
- Insertar una secuencia de N elementos ordenados de mayor a menor en un montículo de mínimos tiene coste O(N log N), pues cada elemento insertado es el menor y hace falta flotarlo hasta la raíz.
- Para averiguar qué valores pueden haber sido los últimos en insertarse en un montículo, hace falta fijarse en la raíz de los árboles hermanos, pues es el factor limitante.
- En un montículo de mínimos, el hijo derecho puede tener valores menores que el izquierdo o viceversa, por lo que un recorrido en anchura no necesariamente produciria una secuencia creciente de valores.

SEMANA 3: HEAPSORT Y COLAS DE PRIORIDAD VARIABLE

ALGORITMO HEAPSORT

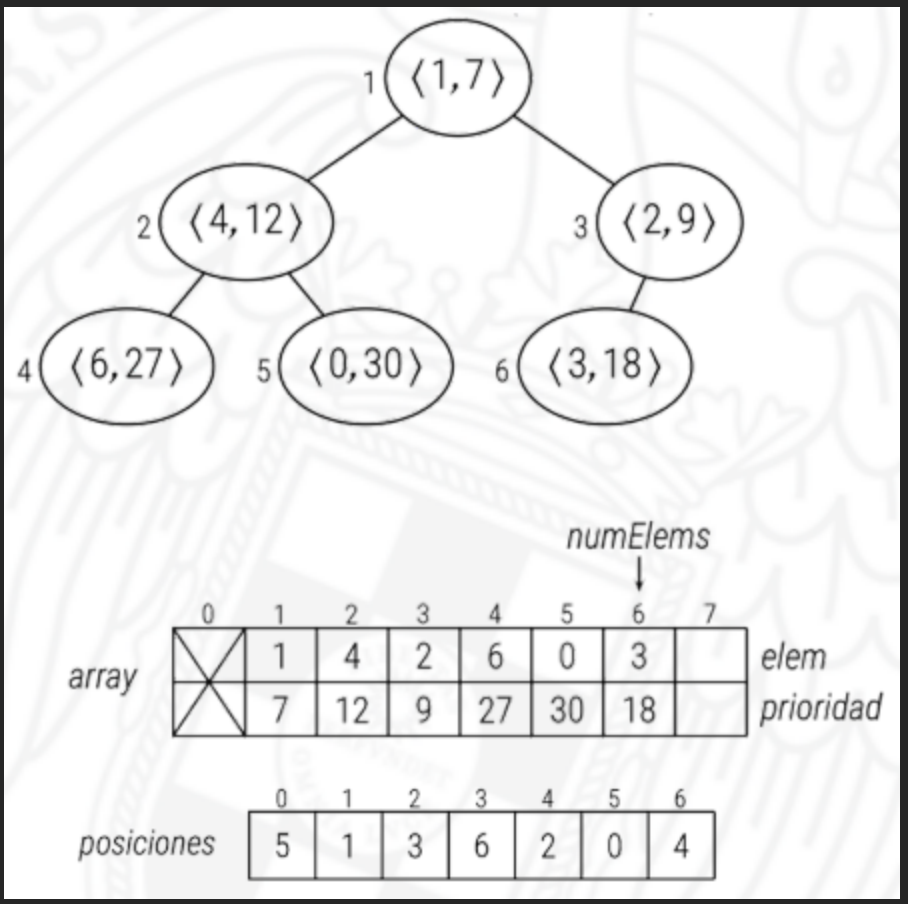
- Implementado con una cola de prioridad, el coste en **tiempo** es de **O(N log N)** y en **espacio adicional** de **O(N)**.
- Para ahorrar dicho espacio adicional:
  1. Se convierte el vector en un montículo (coste: O(N)).
  2. Se extraen sucesivamente los elementos más prioritarios para colocarlos al final del vector.

- Si queremos ordenar el vector de menor a mayor, el montículo deberá ser de máximos y viceversa.

COLAS CON PRIORIDADES VARIABLES

- Permiten cambiar la prioridad de elementos que ya están en la cola.
- Asocian un índice a cada elemento.

Representación



TESTS

- Un vector puede convertirse en un montículo de dos maneras:
  - Recorriendo los elementos de izquierda a derecha y flotando cada uno entre los ya procesados.
  - Recorriendo la primera mitad de derecha a izquierda y hundiendo cada uno entre los siguientes.
- Para ordenar un vector de mayor a menor utilizando heapsort, éste debe convertirse en un montículo de mínimos y viceversa.
- La **complejidad del algoritmo heapsort en espacio adicional es O(1)**, pues todas las transformaciones se realizan sobre el propio vector.
- A la hora de convertir un vector en un montículo para ordenarlo de menor a mayor, el caso peor es cuando el vector está ordenado de menor a mayor, pues hay que convertirlo en un montículo de máximos. Y viceversa.
- La complejidad de convertir un vector de N elementos en un montículo es de O(N), pues se recorre la primera mitad de derecha a izquierda y cada elemento se hunde entre los siguientes.
- El coste en el caso peor de **cambiar la prioridad de un elemento** es de **O(log N)**, pues éste podrá ser flotado o hundido tantas veces como altura tenga el montículo.

SEMANA 4: GRAFOS NO DIRIGIDOS Y SUS RECORRIDOS

TERMINOLOGÍA:

- **Camino**: sucesión de vértices.
- **Ciclo**: camino que empieza y termina en el mismo vértice.
- **Componente conexa**: subgrafo maximal en el que todo par de vértices estan conectados. Si un grafo tiene una única componente conexa, se dice que es conexo.
- **Ciclo euleriano**: ciclo que pasa por todas las aristas una única vez.
- **Ciclo hamiltoniano**: ciclo que pasa por todos los vértices una única vez.
- **Grafo bipartito**: grafo repartido en dos conjuntos disjuntos de tal forma que todas las aristas van de un vértice de un conjunto a un vértice del otro.
- **Grafo planar**: grafo que puede dibujarse en un plano sin que las aristas se crucen.
- **Grafo isomorfo**: grafo que posee biyección de todos sus vértices y aristas (tener en cuenta el grado también).
- **Árbol libre**: grafo cuyo todo par de vértices está conectado por un único camino.
- **Bosque**: grafo cuyas componentes conexas son todas árboles libres.

Representación

representación	espacio	añadir arista $v - w$	comprobar si v y w son adyacentes	recorrer los vértices adyacentes a v
matriz de adyacencia	V^2	1	1	V
listas de adyacentes	$V + A$	1	grado(v)	grado(v)
conjuntos de adyacentes	$V + A$	log V	log V	grado(v)
lista de aristas	A	1	A	A

Método de resolución de problemas de grafos:

1. Crear una clase específica para cada problema.
2. Implementar su constructora, la cual realizará cierto trabajo sobre el grafo.
3. El usuario creará un grafo.

4. El usuario creará un objeto de la clase "Problema", pasándole el grafo como argumento.
5. Ahora podrá utilizar los métodos de consulta de la clase para averiguar propiedades del grafo.

Recorrido en profundidad

- Encontrar los vértices alcanzables desde uno dado (resolución de un laberinto).
- Coste: **O(V + A)**
- Visitar un vértice consiste en:
  1. Marcarlo como visitado.
  2. Hacer algo con él.
  3. Visitar recursivamente todos sus vértices adyacentes aún no visitados.

Recorrido en anchura

- Encontrar el camino más corto entre dos vértices.
- Visita todos los vértices alcanzables a distancia 1, luego a distancia 2, y asi sucesivamente.
- Coste: **O(V + A)**
- Tendremos 3 vectores:
  1. Visitado
  2. Anterior
  3. Distancia

TESTS

- El **número máximo de aristas** que puede tener un grafo no dirigido de V vértices es: **V ∗ (V - 1) / 2**, puesto que existiría una arista entre cada par de vértices.
- En un recorrido en anchura, el mínimo número de veces que se añade cada grafo a la cola es 0, pues los vértices no alcanzables desde el origen no se añaden nunca.
- Si empleamos listas de adyacentes para implementar un grafo, cada arista da lugar a dos nodos.
- El coste de un **algoritmo que calcula el grado de cada vértice** de un grafo implementado mediante una **matriz de adyacencia** es de **O(V^2)**.
- En un recorrido en anchura, la cola puede contener algunos vértices de el nivel actual y algunos del siguiente.
- Un grafo no dirigido y conexo tiene como **mínimo**: **V - 1 aristas**.
- La complejidad del **algoritmo que calcula el número de aristas de un grafo** implementado mediante una **matriz de adyacencia** tiene coste **O(V^2)**.
- La complejidad de un a**lgoritmo que calcula el grado de cada vértice** de un grafo implementado mediante una **lista de adyacentes** tiene coste **O(V)**, puesto que no hace falta recorrer cada lista sino acceder a su longitud.
- Para calcular los árboles de un bosque de V vértices y A aristas: V - A.

- El coste de averiguar si dos vértices son adyacentes en un grafo implementado mediante una lista de adyacentes tiene coste O(V), puesto que las listas están acotadas por el número de vértices.
- El coste de un **recorrido en anchura** de un grafo implementado mediante una **matriz de adyacencia** es de **$O(V^2)$**.

SEMANA 5: GRAFOS DIRIGIDOS Y SUS RECORRIDOS

- **Digrafo**: grafo con aristas dirigidas.
- Caminos y ciclos funcionan de la misma manera, solo que con aristas dirigidas.
- Componente conexa -> Componente fuertemente conexa
- **Grafo inverso o traspuesto**: mismos vértices pero aristas dadas la vuelta.
- **Grafo complementario**: grafo que contiene todas las aristas que no tenía el original y viceversa.

Tipos de problemas nuevos:

- Ordenación topológica: ordenar los vértices de forma que todas las aristas apunten en el mismo sentido.
- Cierre transitivo: encontrar todos los vértices conectados.
- PageRank: calcular importancia (grado de entrada) de una página web (vértice).

Representación

representación	espacio	añadir arista $v \rightarrow w$	comprobar si v y w son adyacentes	recorrer los vértices adyacentes a v
matriz de adyacencia	V^2	1	1	V
listas de adyacentes	$V + A$	1	grado-sal(v)	grado-sal(v)
lista de aristas	A	1	A	A

Matriz de adyacencia:

- Grado de salida: recorrer por filas
- Grado de entrada: recorrer por columnas

Listas de adyacentes:

- Grado de salida: consultar su array
- Grado de entrada: calcular el grafo inverso y consultar su array.

Recorrido en profundidad: mismo funcionamiento y coste que en grafos no dirigidos.
Recorrido en anchura: mismo funcionamiento y coste que en grafos no dirigidos.

ORDENACIÓN TOPOLÓGICA

- No puede tener ciclos.
- Postorden: añadir el vértice al orden tras la llamada recursiva (recorrido en profundidad).
- Postorden inverso: ordenación topológica.
- Demostración: para toda llamada dfs(v)
  1. dfs(w) empezó y terminó
  2. dfs(w) aún no ha empezado.
  3. dfs(w) empezo pero no ha terminado. <- IMPOSIBLE: supondría un ciclo entre v y w.

Detección de ciclos

- Al realizar un recorrido en profundidad, si llegamos a un vértice no solo visitado, si no tambíen apilado, quiere decir que existe un ciclo.
- Debemos hacer uso de un nuevo vector de apilados.
- Al principio de la función recursiva el vértice se marca como apilado y, al final, como desapilado.

TESTS

- Pueden haber tantas componentes conexas como vértices en un grafo dirigido, puesto que puede no tener aristas.
- Para garantizar que los vértices se visitan por distancias en un recorrido en anchura, se utiliza una cola (no una pila).
- El coste de calcular el grado de salida o entrada de cada uno de los vértices de un grafo dirigido implementado mediante una **matriz de adyacencia** es de $\mathbf{O(V^2)}$, pues hay que recorrerla entera.
- El coste de **calcular el grafo traspuesto** de un grafo dirigido implementado mediante **listas de adyacencia** es de **O(V + A)** pues hay que recorrer todas las listas y, para cada adyacencia, añadir la traspuesta.
- El coste de calcular el **grafo complementario** de un grafo dirigido implementado mediante una **matriz de adyacencia** es de $\mathbf{O(V^2)}$, pues hay que recorrerla entera.
- El coste de calcular el **grafo complementario** de un grafo dirigido implementado mediante **listas de adyacencia** es de $\mathbf{O(V^2)}$ pues, para cada una, hay que tener en cuenta las aristas que existen y las que no.
- Si un grafo no tiene ciclos, el postorden inverso es siempre ordenación topológica.
- Para calcular ordenaciones topológicas:
  1. Iterar las componentes conexas empezando por las raices menores.
  2. Colocar el recorrido en profundidad en orden pero al final.
-

Un grafo dirigido puede tener como máximo V! ordenaciones topológicas, puesto que puede no tener aristas.

- Un **grafo dirigido** puede tener como **máximo V * (V - 1) aristas**.
- Una ordenación topológica tiene los nodos ordenados en función de sus aristas, que unen vértices posteriores en la secuencia.

SEMANA 6: CONJUNTOS DISJUNTOS, GRAFOS VALORADOS Y ARMs

- Se representan mediante árboles.
- Si un elemento es raíz, su posición en el vector coincidirá con su valor.
- Si un elemento es un hijo, su valor en el vector corresponderá al del padre.
- Al unir, el árbol más pequeño pasa a ser hijo del mayor.
- Para mejorar el coste, utilizaremos el método de **compresión de caminos**: durante la búsqueda de la raíz de un hijo, la ráiz de todos sus predecesores se cambiará a ésta para así disminuir la profundidad el árbol. Coste: **O(N + M lg* N)**.

M llamadas a unir y buscar sobre una partición con N elementos

Implementación	complejidad en el caso peor
búsqueda rápida	$N\ M$
unión rápida	$N\ M$
unión rápida por tamaños	$N + M \log N$
unión rápida con compresión de caminos	$N + M \log N$
unión rápida por tamaños y con compresión de caminos	$N + M \lg^* N$

ÁRBOLES DE RECUBRIMIENTO

Un árbol es de recubrimiento cuando:

-

Es conexo
- Es acíclico
- Alcanza todos los vértices

Propiedades
- Tienen V - 1 aristas.
- Al eliminar cualquier arista, deja de ser conexo.
- Añadir cualquier arista crea un ciclo.
- **Propiedad del corte**: dados dos subconjuntos no vacíos de vértices, la arista de menor peso que cruce entre ellos pertence al ARM.

Algortimo de Kruskal
Calula un ARM.
1. Seleccionar en cada iteración la arista de menor peso que no haya sido elegida y que no forme ciclo.
2. Incluir aristas hasta que el grafo parcial sea conexo.

Si el grafo no es conexo se calcula un bosque de recubrimiento mínimo.

Operación	Frecuencia	Coste por operación
construir cola prioridad	1	A
construir partición	1	V
`pop`	A	$\log A$
`unir`	$V - 1$	$\lg^* V$
`unidos`	A	$\lg^* V$

TESTS
- Para averiguar qué vectores podrían ser el resultado de realizar uniones rápidas: considerar los tamaños de los subárboles.
- Un grafo no dirigido completo con V > 2 vértices y todas las aristas con valor 1 tiene múltiples ARMs con coste V - 1.

- El coste de **unir** conjuntos disjuntos implementados con **búsqueda rápida** es de **O(N)**, pues hay que recorrer todo el vector para mover los elementos de un conjunto a otro.
- El número máximo de uniones que se pueden realizar sobre una partición de N elementos es N - 1.
- El coste de **buscar o unir** conjuntos disjuntos implementados con **unión rápida por tamaños** es de **O(log N)**, pues buscar la raíz tiene un coste equivalente a la altura del subárbol.
- El coste de **buscar** conjuntos disjuntos implementados con **unión rápida** es de **O(N)**, pues la altura de un árbol puede llegar a ser N.
- Los ARMs no consideran puntos de partida al ser no dirigidos.
- Un árbol de recubrimiento de un grafo conexo con V vértices tiene V + 1 aristas.
- Un **grafo conexo pero cíclico** tiene **varios árboles de recubrimiento**, uno acíclico no.
- El orden de complejidad de la altura de una estructura implementada con unión rápida por tamaño y compresión de caminos es de O(1), pues todos los árboles tienen altura 1 o 2.
- Si todos los valores de las aristas son distintos, el ARM es único.
- Todo ARM contiene a la arista de menor valor.
- Si un ARM contiene a la arista de mayor valor, ésta es imprescindible para hacerlo conexo.
- El coste del algoritmo que calcula si un grafo implementado con unión rápida por tamaño y compresión de caminos es acíclico es de O(V + A lg* V), pues se realizan A operaciones de unión sobre V elementos.

SEMANA 7: DIGRAFOS VALORADOS Y CAMINOS MÍNIMOS

Variantes del problema de camino único:

- **Origen único**: de un vértice a todos los demás.
- **Destino único**: de cualquier vértice a uno concreto.
- **De punto a punto**: de un vértice a otro.
- **Entre cualquier par de vértices**.

Restricciones sobre los pesos:

- **Pesos no negativos**
- **Pesos euclídeos**
- **Pesos arbitrarios**

Presencia de ciclos:

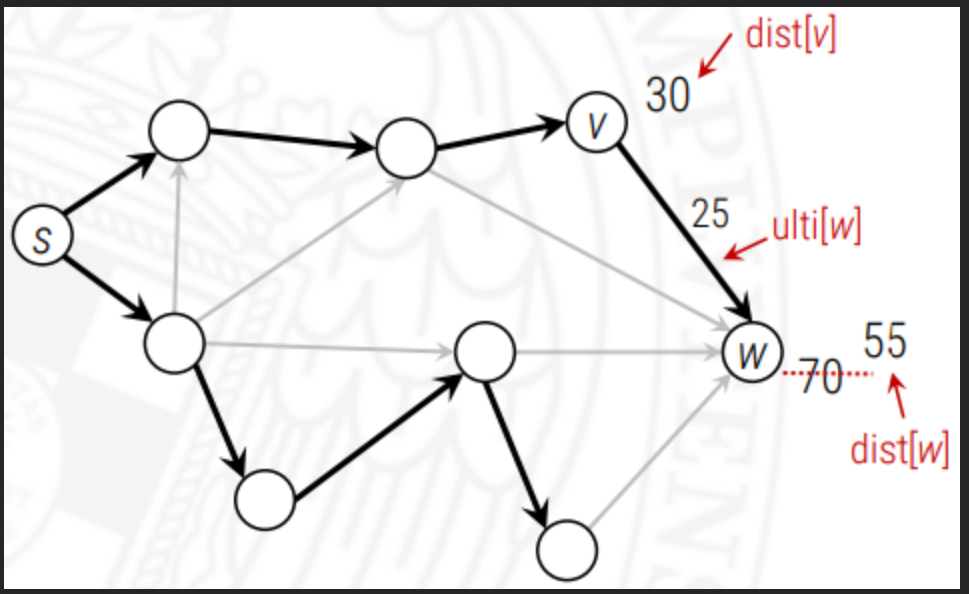
- **Ciclos dirigidos**
- **Sin ciclos dirigidos**
- **Sin ciclos de coste negativo**

El camino mínimo forma un árbol de camino mínimo.
Se utilizan dos vectores:

1. dist[v]: distancia entre el origen y un vértice v.
2. ulti[v]: última arista del camino más corto entre el origen y v.

Relajación de aristas

Si se encuentra un camino más corto del origen a un vértice cualquiera, se actualizan tanto dist[w] como ulti[w].



Condiciones de optimalidad

1. dist[s] = 0 (distancia al origen).
2. Para todo v, dist[v] es la longitud de algún camino de s a v.
3. Para toda arista v -> w, dist[w] <= dist[v] + el valor de la arista.

Algoritmo de Dijkstra

Camino más corto desde un vértice origen a todos los demás. Coste: **O(A log V)** y en espacio adicional de **O(V)**.

1. Se consideran los vértices en orden creciente de distancia desde el origen.
2. En cada iteración, se añade al árbol y se relaja todas las aristas que salen de él.

Operación	Frecuencia	Coste por operación
inicializar los vectores	1	V
construir cola prioridad	1	V
`pop`	V	$\log V$
`update`	A	$\log V$

TESTS

- Cada vértice se inserta y elimina de la cola como máximo 1 vez.
- El coste de calcular el camino más corto entre **todos los pares de vértices** es de **O(V * A log V)**.
- El coste del camino mínimo queda fijado cuando el vértice sale de la cola de prioridad, y los vértices van saliendo de la cola por orden creciente de costes.
- El número mínimo de veces que se invocan las operaciones de insertar y borar el mínimo de la cola por cada vértice es 0, pues los no alcanzables no se insertan nunca.
- El número máximo de veces que se invocan las operaciones de insertar y borar el mínimo de la cola por cada vértice es 1, aunque su prioridad pueda cambiar varias veces.
- El orden de complejidad del número total de veces que se realiza la operación que decrece la prioridad de un elemento de la cola es de O(A), puesto que se invoca con todas las aristas del grafo.
- En un grafo conexo no dirigido con todos los valores de las aristas distintos y positivos, si el valor de todas las aristas se incrementa en la misma cantidad, el **ARM no cambia** pero el **camino más corto entre cada par de vértices puede hacerlo**.
- Para toda arista v -> w, dist[w] <= dist[v] + el valor de la arista.
- Los grafos con aristas de valores negativos tienen caminos de coste mínimo si no contienen ciclos de coste negativo (algoritmo de Bellman-Ford).
- Las aristas se relajan en orden de su distancia.
- Un grafo dirigido con todos los valores de sus aristas distintos puede tener más de un camino mínimo entre cada par de vértices.

SEMANA 8: ALGORITMOS VORACES (I)

El método voraz construye una solución a través de una serie de "elecciones", las cuales deben ser:

- **Factibles**: satisfacen las restricciones del problema.
- **Óptimas localmente**: la mejor opción local disponible en ese paso.
- **Irrevocables**: no se pueden cambiar en pasos posteriores del algoritmo.

Para construir la solución se dispone de un **conjunto de candidatos**, los cuales van formando dos conjuntos:

- **Seleccionados**: formarán parte de la solución.
- **Rechazados**: definitivamente.

Contamos con:

- **Función de selección**: indica cuál es el candidato más prometedor.
- **Test de factibilidad**: comprueba si un candidato es compatible con la solución parcial.
- **Test de solución**: determina si una solución parcial es completa.
- **Función objetivo**: asocia un valor a cada solución con el fín de encontrar una óptima.

```
fun voraz(datos : conjunto) dev S : conjunto
var candidatos : conjunto
    S := ∅     { en S se va construyendo la solución }
    candidatos := datos
    mientras candidatos ≠ ∅ ∧ ¬es-solución?(S) hacer
        x := seleccionar(candidatos)
        candidatos := candidatos – {x}
        si es-factible?(S ∪ {x}) entonces S := S ∪ {x} fsi
    fmientras
ffun
```

Método de reducción de diferencias

- Compara una solución óptima con la solución obtenida.
- Si no son iguales, se va transformando la óptima inicial de forma que continúe siendo óptima pero mas parecida a la obtenida.

TESTS

-

En el problema de la mochila real, puede haber varias soluciones óptimas con distinto número de objetos enteros. Es decir, depende del valor de los mismos.

- En el problema de la mochila real, la solución óptima obtenida no tiene porqué siempre contener una fracción de un objeto, puede haber un subconjunto de objetos que rellenen de forma exacta la mochila.
- El algoritmo voraz no resuelve de forma óptima el problema de la mochila real sin fraccionar.
- El algoritmo voraz que resuelve el problema del cambio de monedas no siempre produce soluciones óptimas.
- En el problema de la mochila real, **el peso límite no influye en el coste asintótico del algoritmo**. Éste viene dado por el coste de ordenar los objetos de mayor a menor valor por unidad de peso, que está en **O(N log N)**. La segunda fase, de selección, los recorre todos en el caso peor y tiene coste O(N).
- Los algoritmos de Kruskal y Dijkstra son voraces.
  - **Kruskal**: encuentra el ARM seleccionando las aristas por orden creciente de coste.
  - **Dijkstra**: calcula los caminos mínimos considerando los vértices en orden creciente de distancia al origen.
- El coste del algoritmo voraz que resuelve el problema de la mochila real es O(N log N).

SEMANA 9: ALGORITMOS VORACES (II)

Tiempo en el sistema mínimo

- Tenemos N tareas y queremos minimizar el tiempo medio de estancia de una tarea en el sistema.
- Atiende a las tareas por orden creciente de tiempo de ejecución.

Tareas con plazo y beneficio

- Tenemos N tareas con un plazo *p* y un beneficio *b*.
- El objetivo es maximizar el beneficio total obtenido.
- Considera las tareas de mayor a menor beneficio y las selecciona si es factible.

Demostración de optimalidad

1. Transformar las secuencias de forma que las tareas comunes se realicén en el mismo momento.
2. Comparar las secuencias, las cuales pueden diferir de tres maneras:
   1. **X realiza una tarea mientras que Y libra**: imposible, puesto que Y es óptima y la hubiese seleccionado también.
   2. **Y realiza una tarea mientras que X libra**: imposible, si X rechazó dicha tarea, no puede seleccionarla más tarde.
   3. **X e Y realizan tareas distintas**, las cuales deben tener el mismo beneficio por haber seguido la estrategia voraz.

Test de factibilidad: se planifican las tareas lo más tarde posible, respetando su plazo. Coste: O(N log* N).

SEMANA 10: PROGRAMACIÓN DINÁMICA (I)

- Se utiliza una tabla para almacenar los resultados de subproblemas ya resueltos.
- La tabla tiene tantas dimensiones como argumentos tiene la recurrencia.
- El tamaño de cada dimensión coincide con los valores que puede tomar su respectivo argumento.
- Cada subproblema se asocia a una posición de la tabla.

Programación dinámica descendente

- Mantiene el diseño recursivo, es decir, de arriba hacia abajo.
- La función recibe como parámetro la tabla de soluciones, la cual consulta antes de resolver un subproblema.
  - Si este no ha sido resuelto, lo resuelve y almacena la solución.
- Tiene la necesidad de saber si un subproblema está resuelto o no.

Programación dinámica ascendente

- Recorre los problemas de menor a mayor tamaño, es decir, de abajo hacia arriba.
- Todos los subproblemas de tamaño menor deben ser resueltos antes que uno de tamaño mayor.
- La matriz es una variable local de la función.
- Hay que resolver todos los subproblemas, aunque no se usen.
- Coste: **O(N*R)** y en espacio adicional de **O(R)**.

PROBLEMA DEL CAMBIO DE MONEDAS

- Tenemos un conjunto finito M de tipos de monedas.
- Existe una cantidad ilimitada de monedas de cada valor.
- Se quiere pagar una cantidad C utilizando el menor número de monedas.

- Casos recursivos:

$$monedas(i,j) = \begin{cases} monedas(i-1,j) & \text{si } m_i > j \\ \min(monedas(i-1,j), monedas(i,j-m_i)+1) & \text{si } m_i \leq j \end{cases}$$

donde $1 \leq i \leq n$ y $1 \leq j \leq C$

- Casos básicos:

$$monedas(i,0) = 0 \qquad 0 \leq i \leq n$$
$$monedas(0,j) = +\infty \qquad 1 \leq j \leq C$$

- Llamada inicial: $monedas(n, C)$

Principio de optimalidad de Bellman

- Para conseguir una solución óptima basta con considerar subsoluciones óptimas.
- Se cumple en un problema si una solución óptima a una instancia del problema siempre contiene soluciones óptimas a todas sus subinstancias.

TESTS

- El número de subproblemas distintos que se resuelven para calcular un número combinatorio utilizando programación dinámica **descendente** viene dado por una subsección rectangular del triángulo de Tartaglia, cuyos vértices son: (n | r), (n-r | 0), (r | r) y (0 | 0).
- El **número de veces que se repite cada subproblema** en programación dinámica **ascendente** esta en el orden de **O(1)**.
- En programación dinámica **ascendente, cada subproblema se consulta a lo sumo dos veces**, para calcular los términos (i+1 | j) y (i+1 | j+1).
- Al calcular un número combinatorio utilizando **recursión sin memoria**, el número de ves que se repite un problema viene dado por (n-2 | r-1), que está en el **orden de O(2^n)**.
- El coste del algoritmo de resuelve el problema del **cambio de monedas** con programación dinámica **ascendente** es de **O(N * R)**.
- El problema del cambio de monedas puede tener varias soluciones óptimas.
-

La cantidad de memoria adicional para una función recursiva que depende únicamente de su valor anterior es O(1), pues basta con utilizar una sóla variable.

- El número de subproblemas distintos que se resuelven para calcular un número combinatorio utilizando programación dinámica **ascendente** es (n + 1)* (r + 1).

SEMANA 11: PROGRAMACIÓN DINÁMICA (II)

PROBLEMA DE LA MOCHILA ENTERA

- No importa el orden, solo los objetos disponibles y el peso máximo de la mochila.
- Puesto que los objetos no son ilimitados, se eliminan en su llamada recursiva (i - 1).

- Casos recursivos:

$$mochila(i,j) = \begin{cases} mochila(i-1,j) & \text{si } p_i > j \\ \max(mochila(i-1,j), mochila(i-1,j-p_i)+v_i) & \text{si } p_i \le j \end{cases}$$

con $1 \le i \le n$ y $1 \le j \le M$

- Casos básicos:

$$mochila(0,j) = 0 \qquad 0 \le j \le M$$
$$mochila(i,0) = 0 \qquad 0 \le i \le n$$

- Llamada inicial: $mochila(n, M)$

TIRO AL PATÍNDROMO

- Consiste en conseguir el palíndromo más largo tirando algunos de los patitos (si fuese necesario).

- Casos recursivos $(i < j)$:

$$patíndromo(i,j) = \begin{cases} patíndromo(i+1,j-1)+2 & \text{si } patitos[i] = patitos[j] \\ \text{máx}(\ patíndromo(i+1,j), & \\ \quad patíndromo(i,j-1)\ ) & \text{si } patitos[i] \neq patitos[j] \end{cases}$$

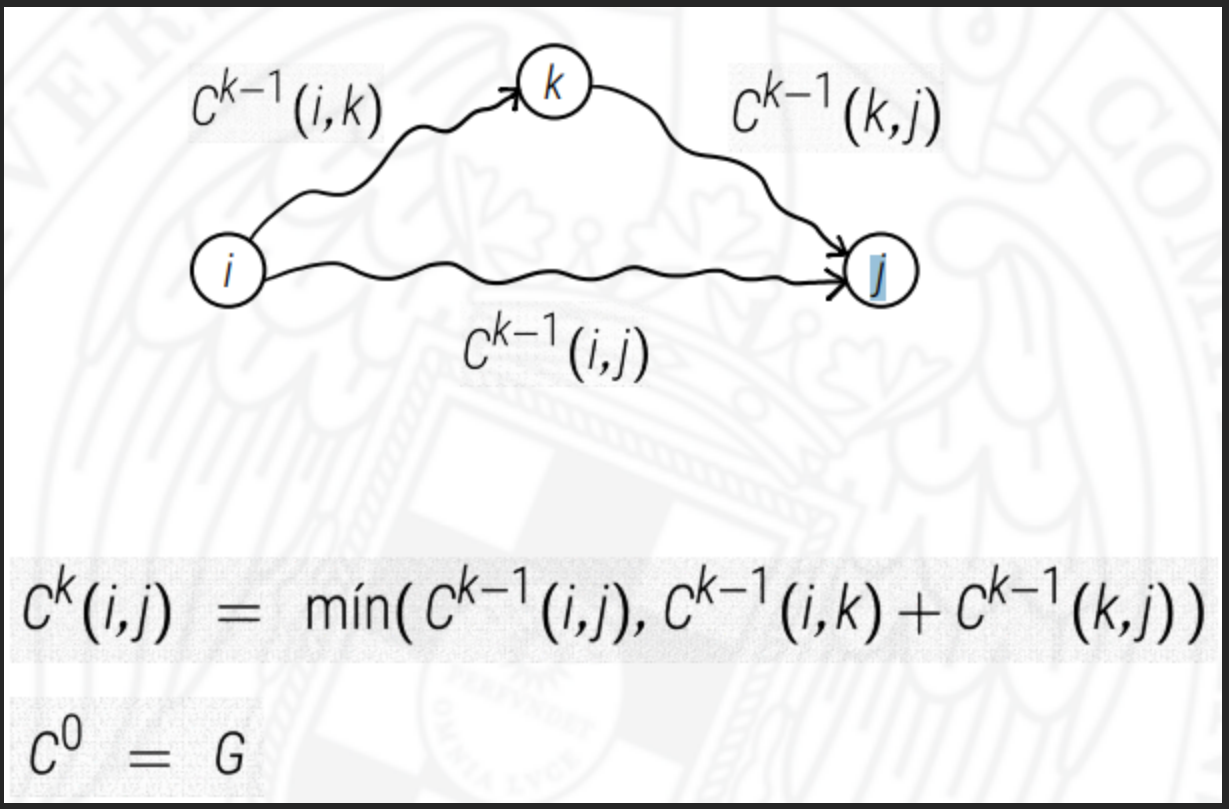
- Casos básicos:

$$\begin{aligned} patíndromo(i,i) &= 1 \\ patíndromo(i,j) &= 0 \quad \text{si } i > j \end{aligned}$$

- Llamada inicial: $patíndromo(0, n-1)$

CAMINOS MÍNIMOS ENTRE TODO PAR DE VÉRTICES

- Dado un drigrafo valorado, calcular el camino de coste mínimo entre cada par de vértices.
  - Si los pesos son positivos y el grafo disperso, podemos utilizar el algoritmo de Dijkstra V veces, con un coste en O(V A log V).
  - El algoritmo de Floyd resuelve el caso general, con pesos posiblemente negativos y coste $O(V^3)$.



$$C^k(i,j) = \min(C^{k-1}(i,j), C^{k-1}(i,k) + C^{k-1}(k,j))$$

$$C^0 = G$$

TESTS

- La **capacidad de la mochila influye en el coste asintótico del algoritmo de programación dinámica**, pues su coste está en O(N * M), siendo N el número de objetos y M el peso límite.
- El problema de la mochila tiene un coste en espacio adicional de O(N * M), pues para obtener los objetos seleccionados son necesarias las celdas pertenecientes a filas anteriores.
- El problema de la mochila resuelto con programación dinámica utiliza valores enteros para los pesos, pues es necesario para indexar la tabla.
- En el problema de los caminos mínimos, la matriz se puede rellenar de cualquier forma, puesto que en la iteración k-ésima **los valores de la fila y columna *k* no se modifican**.
- El coste de la resconstrucción de todos los caminos en el algoritmo de Floyd tiene coste $O(V^3)$, pues hay $O(V^2)$ parejas de vértices y la reconstrucción de cada uno está en O(V).
- En el algoritmo de Floyd con grafos sin ciclos de coste negativo, los elementos de la diagonal siempre valen 0.
- El algoritmo de Floyd se puede utilizar en grafos con aristas de coste negativo siempre y cuando **no haya ciclos de coste negativo**.
- El principio de optimalidad de Bellman garantiza que para encontrar una solución óptima basta con considerar las soluciones óptimas de los subproblemas, pero eso no evita tener que explorar varios subproblemas correspondientes a diferentes elecciones para determinar cuál es la mejor.
-

En el problema de la mochila con reducción del coste en espacio, el vector se debe rellenar de **derecha a izquierda** para mantener los valores disponibles hasta realizar el cálculo.

- Si queremos reducir el coste en espacio del problema de la mochila, no podemos devolver los objetos que forman parte de la solución óptima.
- El algoritmo de Floyd y Dijkstra resuelven distintos problemas:
  - **Floyd**: calcula los costes de los caminos mínimos entre todos los pares de vértices.
  - **Dijkstra**: resuelve el problema de calcular los costes de los caminos mínimos de un sólo vértice a todos los demás. Se podría invocar V veces para resolver el problema como Floyd.
- El coste del algoritmo de Floyd es independiente del número de aristas, por lo que da igual si el grafo es denso o disperso.
- El algoritmo de Floyd se puede utilizar para detectar la **existencia de ciclos de coste negativo**.
- Para reconstuir los caminos mínimos en el algoritmo de Floyd, son necesarias todas las matrices anteriores y no solo la última.
- El algoritmo de Floyd también calcula el coste de los caminos mínimos entre un vértice y todos los demás, pero con un coste mayor que Dijkstra.

SEMANA 12: PROGRAMACIÓN DINÁMICA (III)

MULTIPLICACIÓN ENCADENADA DE MATRICES

- El producto de una matriz A_{pxq} y B_{qxr} es una matriz C_{pxr}
- Se necesitan PQR multiplicaciones
- La diagonal principal se inicializa con 0s.
- Tendremos n - 1 diagonales con n - d elementos.
- Para acceder a la matriz:
  - La fila coincide con el número del elemento dentro de la diagonal.
  - La columna corresponde con i + d.
- La matriz secundaria P guarda los valores de k que dieron el menor resultado para un determinado (i, j).

$$\underbrace{(M_1 \cdot \ldots \cdot M_k)}_{d_0 \times d_k} \cdot \underbrace{(M_{k+1} \cdot \ldots \cdot M_n)}_{d_k \times d_n}$$

$matrices(i,j)$ = número *mínimo* de multiplicaciones básicas para realizar el producto matricial $M_i \cdots M_j$

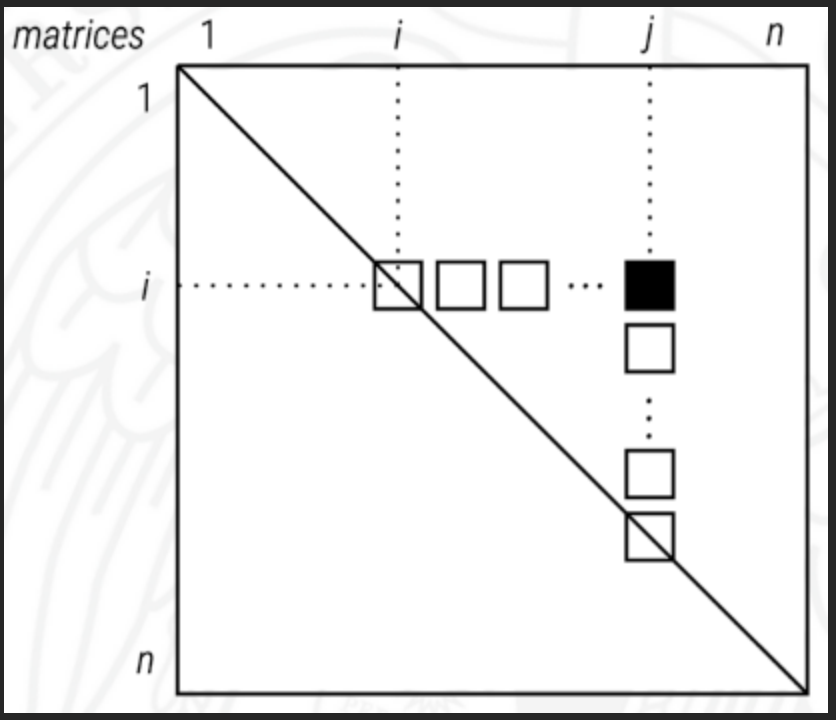
- Casos recursivos, $i < j$.

$$matrices(i,j) = \text{mín}_{i \le k \le j-1} \{matrices(i,k) + matrices(k+1,j) + d_{i-1} d_k d_j\}$$

- Casos básicos:

$$matrices(i,i) = 0$$

- Llamada inicial: $matrices(1, n)$



JUSTIFICACIÓN DE UN TEXTO

- Dadas N palabras y sus longitudes, distribuirlas en un párrafo con lineas de longitud L.

Solución 1

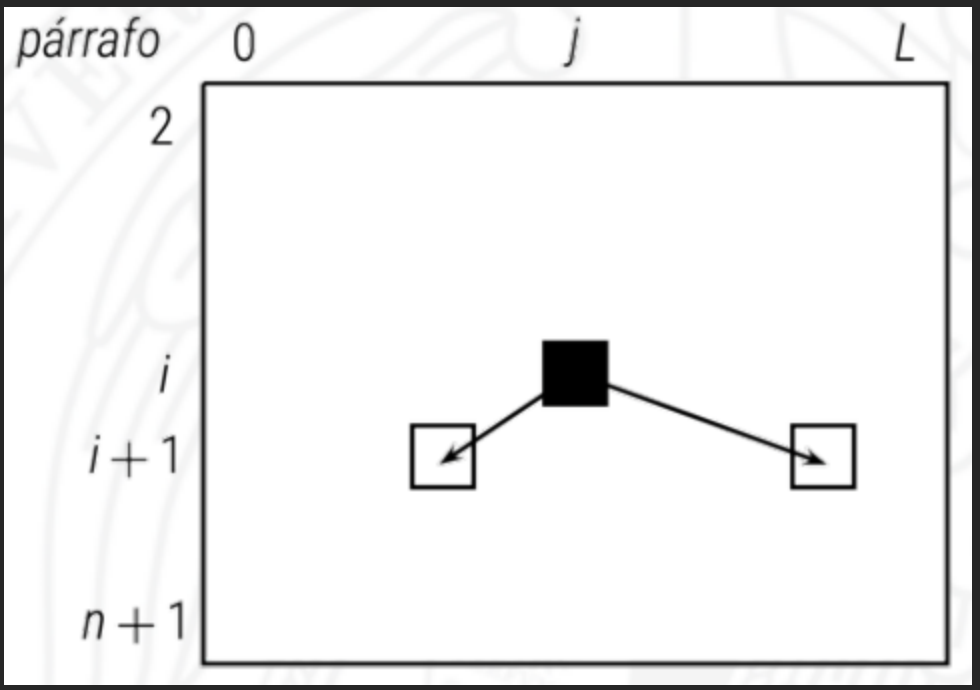
- Casos recursivos, $i \leq n$

$$párrafo(i,j) = \begin{cases} párrafo(i+1, L-l_i) + j^3 & \text{si } l_i + 1 > j \\ \min(párrafo(i+1, L-l_i) + j^3, & \\ \quad párrafo(i+1, j-(l_i+1))) & \text{si } l_i + 1 \leq j \end{cases}$$

- Casos básicos:

$$párrafo(n+1, j) = 0$$

- Llamada inicial: $párrafo(2, L - l_1)$



Solución 2

$párrafo(i)$ = penalización *mínima* al formatear las palabras de la i a la n empezando en una línea en blanco

$$párrafo(i) = 0 \quad \text{si } caben(i,n)$$

$$párrafo(i) = \min_{\substack{i \le j < n \\ caben(i,j)}} \{penaliza(i,j) + párrafo(j+1)\}$$

donde $caben(i,j) = (j-i) + \sum_{k=i}^{j} l_k \le L$

TESTS

- El coste del problema de la **multiplicación encadenada** es de **O(N^3)**.
- El problema de la multiplicación encadenada puede implementarse utilizando el método recursivo descendente, en el cual la tabla almacenará los valores de los subproblemas necesarios para calcular el resultado del problema original.
- El coste del problema de la multiplicación encadenada con reducción del coste (matriz de paréntesis) en espacio tiene coste O(N) en lugar de O(N^2).
- En el problema de la justificación de un texto con un argumento, el número de casos base depende tanto de la longitud de la línea como de las longitudes de las palabras.
- Para reconstruir la solución del problema de la justificación de un texto, basta con un espacio adicional en O(N).
- En el problema de la justificación de un texto con un argumento, las posiciones de la tabla correspondientes a los casos base son las últimas, pues son las que cumplen que todas las palabras desde la i hasta la última caben en la línea.
- No hay una estrategia voraz óptima que resuelva el problema de la justificación de un texto.

SEMANA 13: RAMIFICACIÓN Y PODA Y ÁRBOLES DE JUEGO

- No siempre es posible utilizar divide y vencerás, voraces o programación dinámica para lograr soluciones eficientes, pues es impracticable para un conjunto de soluciones posibles muy grande.
- Una solución debe minimizar, maximizar o satisfacer la **función criterio**.
- Las restricciones pueden ser:
  - **Explícitas**: que indican los conjuntos S.
  - **Implícitas**: relaciones entre los componentes de la solución.
- El **espacio de soluciones** estará formado por el conjunto de tuplas que satisfacen la restricciones explícitas y se estructura como un árbol de exploración.
- Se utilizan **funciones de poda** que permiten determinar cuándo una solución parcial no va a conducir a una solución.
- El algoritmo consiste en realizar un recorrido del árbol de exploración hasta:
  - Encontrar la primera solución.
  - Recorrer el árbol completo para obtener todas las soluciones o una óptima.
- Existen dos tipos de recorridos:
  - **Vuelta atrás**: el recorrido se realiza en profundidad, método sencillo y eficiente en espacio.
  - **Ramificación y poda**: búsqueda más inteligente, se expande el nodo vivo más prometedor (cola de prioridad).

<u>Ramificación y poda</u>

- Dispone de una función de **coste estimado**, que proporciona una cota inferior al coste de la mejor solución alcanzable desde el nodo actual.
- Si su coste estimado se aproxima suficientemente al del coste real, consideramos como más prometedor al nodo vivo con menor coste estimado.

```
fun ramificación-y-poda-mín(T : árbol-de-estados) dev ⟨sol-mejor : tupla, coste-mejor : valor⟩
var X, Y : nodo, C : colapr[nodo]
    Y := raíz(T)
    C := cp-vacía() ; añadir(C, Y)
    coste-mejor := +∞
    mientras ¬es-cp-vacía?(C) ∧ coste-estimado(mínimo(C)) < coste-mejor hacer
        Y := mínimo(C) ; eliminar-mín(C)
        para todo hijo X de Y hacer
            si es-solución?(X) entonces si coste-real(X) < coste-mejor entonces
                                            coste-mejor := coste-real(X) ; sol-mejor := solución(X)
                                        fsi
            si no si es-completable?(X) ∧ coste-estimado(X) < coste-mejor entonces
                        añadir(C, X)
                    fsi
            fsi
        fpara
    fmientras
ffun
```

TAREAS CON PLAZO FIJO, DURACIÓN Y COSTE

- Cada una de las N tareas tiene asociada una terna (T, P, C) con su tiempo, plazo y coste correspondientes.
- El objetivo es seleccionar un conjunto S de las N tareas que puedan realizarse dentro de su plazo y con coste mínimo.
- Un subconjunto S de tareas es factible si y solo si la secuencia que las ordena de forma no decreciente según el plazo es admisible.

PROBLEMA DE LA MOCHILA ENTERA

- Al igual que en versiones anteriores, el problema consiste en maximizar el beneficio sin pasarse del límite de peso.
- Como el problema es de maximización, necesitamos una cota superior del beneficio de la mejor solución alcanzable desde un nodo.

TESTS

- El **coste de los algoritmos de ramificación y poda es proporcional al tamaño del espacio de soluciones**, pues en el caso peor lo explora entero.

El algoritmo de ramificación y poda se puede utilizar para encontrar todas las soluciones que satisfacen un conjunto de restricciones, asignando la misma prioridad a todos los nodos y deteniendo el algoritmo solamente cuando la cola de prioridad está vacía.

- En un problema de ramificación y poda, el número de nodos que podrían generarse son 2^h - 1, y el número de posibles soluciones es el número de nodos del último nivel: 2^{h-1}.
- La técnica de ramificación y poda no sigue ningún patrón fijo como el recorrido en profundidad o en anchura, aunque podrían conseguirse asignando las prioridades adecuadas.
- En un problema de maximización, la estimación utilizada ha de ser una cota superior. De esta forma, si la estimación de un nodo es inferior al valor de la mejor encontrada hasta el momento, se descarta. Viceversa.
- El algoritmo de ramificación y poda puede detenerse:
  - Si la cola de prioridad queda vacía.
  - SI la prioridad del más prioritario es peor que el valor de la mejor solución encontrada, por lo que todos los nodos que permanecen en la cola son no prometedores.

PRÁCTICA

TAD DE CONJUNTOS AVL TreeSet_AVL.h

Implementado mediante nodos, que además de un valor, guardan la altura.

Operaciones:

- **Constructora**: Set
- **Insertar**: bool insert(T const& e) //O(log N)
- **Eliminar**: bool erase(T const& e)
- **Contains**: bool contains(T const& e) const
- **Empty**: bool empty() const
- **Size**: int size() const
- **Iteradores**

Subfunciones:

- **Reequilibrar izquierda**:void reequilibraIzq(Link & a)
- **Reequilibrar derecha**: void reequilibraDer(Link & a)
- **Rotación simple izquierda**: void rotaDer(Link & r2)
- **Rotación simple derecha**: void rotaIzq(Link & r2)
- **Rotación doble izquierda-derecha**: void rotaIzqDer(Link & r3)
- **Rotación doble derecha-izquierda**: void rotaDerIzq(Link & r3)

COLAS DE PRIORIDAD PriorityQueue.h

Podemos cambiar entre una cola de mínimos y de máximos:

```
Máximos: priority_queue<int>
Mínimos: priority_queue<int, vector<int>, greater<int>>
```

Operaciones:

- **Constructora**: ver arriba
- **Insertar**: void push(T const& e)
- **Eliminar primer elemento**: void pop()
- **Top**: T const& top() const
- **Empty**: bool empty() const
- **Size**: int size() const

Subfunciones:

- **Flotar un elemento**: void flotar(int i)
- **Hundir un elemento**: void hundir(int i)

HEAPSORT Diapositivas - Heapsort

Operaciones:

- **Ordenar:** void heapsort(vector & v, Comparador cmp)

Subfunciones:

- **Hundir máximo**: void hundir_max(vector & v, int N, int j, Comparador cmp)

COLAS CON PRIORIDADES VARIABLES IndexPQ.h

Operaciones:

- **Constructora**: IndexPQ(int N)
- **Insertar**: void push(int e, T const& p) -> **e** es el índice, **T** es el tipo de prioridad y **p** dicha prioridad.
- **Modificar prioridad**: void update(int e, T const& p)
- **Top**: Par const& top() const -> devuelve un par de su identificador y su prioridad.
- **Pop**: void pop()
- **Empty**: bool empty() const

- **Size**: int size() const

Subfunciones:
- **Flotar**: void flotar(int i)
- **Hundir**: void hundir(int i)

GRAFOS NO DIRIGIDOS Grafo.h

Operaciones:
- **Constructora**: Grafo(int V)
- **Añadir arista**: void ponArista(int v, int w)
- **Consultar adyacentes a un vértice**: Adys ady(int v) const
- **Num de vértices**: int V() const
- **Num de aristas**: int A() const

Diapositivas - Recorrido en profundidad de grafos no dirigidos

Operaciones:
- **Constructora**: CaminosDFS(Grafo const& g, int s)
- **Comprobar si hay camino del origen a v**: bool hayCamino(int v)
- **Camino desde el origen a v**: Camino camino(int v)

Diapositivas - Máxima componente conexa

Operaciones:
- **Constructora**: MaximaCompConexa(Grafo const& g) : visit(g.V(), false)
- **Tamaño máximo de la componente conexa**: int maximo() const
- **Camino desde el origen a v**: Camino camino(int v)

Diapositivas - Recorrido en anchura de grafos no dirigidos

Operaciones:
- **Constructora**: CaminoMasCorto(Grafo const& g, int s)
- **Comprobar si hay camino del origen a v**: bool hayCamino(int v)
- **Consultar número de aristas entre el origen y v**: int distancia(int v)
- **Camino más corto desde el origen a v**: Camino camino(int v)

GRAFOS DIRIGIDOS Digrafo.h

Operaciones:

- **Constructora**: Digrafo(int V)
- **Añadir arista**: void ponArista(int v, int w)
- **Consultar adyacentes**: Adys ady(int v) const
- **Num vértices**: int V() const
- **Num aristas**: int A() const
- **Grafo inverso**: Digrafo inverso() const (coste: O(V + A))

Diapositivas - Grafos dirigidos

Recorrido en profundiad

Operaciones:

- **Constructora**: DFSDirigido(Digrafo const& g, int s)
- **Comprobar si hay camino del origen a v**: bool alcanzable(int v)

Recorrido en anchura

Operaciones:

- **Constructora**: BFSDirigido(Digrafo const& g, int s)
- **Comprobar si hay camino del origen a v**: bool hayCamino(int v)
- **Consultar número de aristas entre el origen y v**: int distancia(int v)
- **Camino más corto desde el origen a v**: Camino camino(int v)

CONJUNTOS DISJUNTOS ConjuntosDisjuntos.h

Operaciones:

- **Constructora**: ConjuntosDisjuntos(int N)
- **Unir dos conjuntos**: void unir(int a, int b)
- **Buscar un elemento**: int buscar(int a) const
- **Consultar si dos elementos están unidos**: bool unidos(int a, int b) const
- **Cardinal**: int cardinal(int a) const
- **Consultar número de conjuntos**: int num_cjtos() const

GRAFOS VALORADOS GrafoValorado.h

Operaciones:

- **Constructora**: GrafoValorado(int V)
- **Añadir arista**: void ponArista(Arista arista)
- **Num vértices**: int V() const
- **Num aristas**: int A() const
- **Consultar adyacentes**: AdysVal const& ady(int v) const
- **Consultar aristas**: vector<Arista<Valor>> aristas() const

Kruskal
Diapositivas - Árboles de recubrimiento de coste mínimo
Operaciones:

- **Constructora**: ARM_Kruskal(GrafoValorado const& g)
- **Consultar valor**: Valor costeARM()
- **Consultar ARM**: vector<Arista<Valor>> ARM()

DIGRAFOS VALORADOS DigrafoValorado.h

Operaciones:

- **Constructora**: DigrafoValorado<Valor>(int V)
- **Añadir arista**: void ponArista(AristaDirigida arista)
- **Num vértices**: int V() const
- **Num aristas**: int A() const
- **Consultar adyacentes**: AdysDirVal const& ady(int v) const
- **Inverso**: DigrafoValorado inverso() const

Dijkstra
Diapositivas - Caminos mínimos
Operaciones:

- **Constructora**: Dijkstra(DigrafoValorado const& g, int orig)
- **Comprobar si hay camino del origen a v**: bool hayCamino(int v)
- **Consultar número de aristas entre el origen y v**: Valor distancia(int v)
- **Camino más corto desde el origen a v**: Camino<Valor> camino(int v)