Programación y Estructura de Datos Avanzadas

- Teoría Exámenes -

TEMA 2

Tablas Hash (Tablas de Dispersión)

Son tablas que permiten el almacenamiento de datos pertenecientes a un dominio potencialmente muy grande de registros, pero del que sólo usamos y referenciamos un número reducido.

Son tablas que permiten un ahorro considerable de memoria, y son eficientes, siempre que el número de registros del subconjunto se mantenga dentro de lo previsto.

Para lograr la reducción del espacio de claves se hace corresponder a cada clave directa, una clave en el subconjunto de índices de la tabla Hash, mediante una función Hash.

Considerando lo anterior, dos o más claves diferentes del dominio inicial que proporcionen la misma clave en el índice de la tabla Hash, es a lo que se le conoce como colisión.

Funciones Hash

Es una aplicación, no necesariamente inyectiva, entre el conjunto dominio de las claves X y el conjunto dominio de direcciones D de la estructura de datos siguiente, la cual asocia una clave con una posición en la tabla Hash.

$$H : X \rightarrow D$$

$$x \rightarrow h(x)$$

Las funciones h(x) deben tener las siguientes propiedades:

- Deben repartir equiprobablemente los valores (los valores h(x) en D deben mantener una distribución de probabilidad de valores de X).
- h(x) se debe poder calcular de manera eficiente, no permitiendo en ningún caso que la complejidad del proceso de búsqueda sea determinada por el cálculo de h(x).
- Si se producen cambios en la clave, aunque sean pequeños, deben suponer cambios significativos en la función hash h(x).

Hay exactamente dos dimensiones sobre las que poder evaluar funciones Hash:

- *Distribución de Datos:* estudiamos de qué manera y mediante qué distribución estadística se distribuyen los valores dentro del conjunto de datos. Esto implica analizar también las distribuciones de probabilidad de las posibles colisiones.
- *Eficiencia:* una función Hash debe ser estable, rápida y determinista y, en general, es aceptado que el valor medio de complejidad debe ser constante o, a lo sumo, de orden logarítmico.

Resolución de Colisiones en las Funciones Hash

En las funciones Hash utilizadas para la ordenación, es muy frecuente que se den casos de colisión de dos claves, en cuyo caso hay que realizar una resolución de dichas colisiones.

En la resolución de colisiones se requiere almacenar los registros que comparten clave de manera que, en caso de colisión, sean accesibles igualmente.

Existen dos métodos principales: Hashing Abierto y Hashing Cerrado.

La elección de uno u otro depende del factor de carga, el cual es un parámetro que determina cuántas posiciones de la tabla se han ocupado y la probabilidad de encontrar una entrada vacía, definiéndose como:

$$\delta = \frac{n}{M}$$

Donde:

- n: Nº de índices ocupados
- M: Tamaño de la Tabla
- Los valores oscilan entre 0 (Tabla vacía) y 1 (Tabla llena).

El factor de carga mide la proporción de la tabla Hash ya ocupada. Si su valor es 1, se puede resolver doblando el tamaño de la tabla mediante Hashing Abierto. En caso contrario (valor 0) se resolverá mediante Hashing Cerrado.

Hashing Abierto (Resolución de Colisiones)

Utiliza estructuras dinámicas externas a la tabla para el almacenamiento de las claves que han generado colisiones. Desde el punto de vista conceptual, el método es válido, pero no lo es si consideramos la eficiencia, siendo por tanto más eficiente el Hashing Cerrado.

Hashing Cerrado (Resolución de Colisiones)

Permite resolver la colisión mediante la búsqueda en ubicaciones alternativas en la misma tabla, hasta que encontramos un sitio libre en la misma, cuando aparece la presencia de un valor que así lo determine, en cuyo caso, se ubica el valor en la posición indicada por la función Hash.

A medida que la tabla se llena, las probabilidades de colisión aumentan significativamente. Los métodos más conocidos para implementar este tipo de resolución de colisiones son:

- **Recorrido Lineal:** a la dirección obtenida por la función Hash h(k) se le añade un incremento lineal que proporciona otra dirección en la tabla (m: tamaño de la tabla):

  $$s(k,i) = (h(k) + i) \bmod m$$

- **Recorrido Cuadrático:** basado en la expresión cuadrática g(k), permite una mayor dispersión de las colisiones por la tabla que en el recorrido lineal, así como proporciona un recorrido completo por la misma. La exploración cuadrática es completa y recorre todos los bloques de la tabla.
- **Recorrido mediante doble Hashing:** utiliza una segunda función Hash, h'(x), como función auxiliar. Para que el método funcione, la función h'(x) debe cumplir:
  - $h'(x) \neq 0$
  - La función h' debe ser distinta de h
  - Los valores de h'(x) deben ser primos relativos de M para que los índices de la tabla se ocupen en su totalidad. Si M es primo, cualquier función puede ser usada como h'.

Montículos

Son un tipo especial de árbol binario, siempre balanceado, que se implementan sobre vectores con las siguientes propiedades:

- Es un árbol balanceado y completo (los nodos internos tienen siempre dos hijos) con la posible excepción de un único nodo cuando el número de elementos es par.
- Cada nodo contiene un valor mayor o igual que el de sus nodos hijos (montículo de máximos), o menor o igual (montículo de mínimos).

A la segunda de estas propiedades se la denomina "Propiedad de Montículo", con la cual se permite tener en la cima (el nodo raíz) del montículo el elemento mayor o menor, si se trata de un montículo de máximos o de mínimos, respectivamente, siendo esta la utilidad fundamental del montículo.

Los nodos de profundidad k están situados en las posiciones 2^k y siguientes del vector hasta la posición $2^{k+1} - 1$.

El nodo padre del elemento en la posición "i" estará en "i/2 (div., entera)". Los nodos hijos al elemento en la posición "i" estarán en "2*i, 2*(i+1)".

El montículo sirve de apoyo a la creación de un algoritmo de ordenación eficiente, conocido este como Heapsort.

Cuando se extrae el primer elemento de un montículo, restaurar la propiedad de montículo tiene un coste O(log n).

Montículo de Máximos

Cada nodo contiene un valor mayor o igual que el de sus nodos hijos.

Montículo de Mínimos

Cada nodo contiene un valor menor o igual que el de sus nodos hijos, disponiendo con él de una estructura de datos en la que, encontrar el valor mínimo, es una operación de coste constante.

Funciones de los Montículos

- Flotar: Intercambiar por el nodo inmediatamente a un nivel superior.
- Hundir: Intercambiar por el nodo hijo de menor valor (montículo de mínimos).
- Insertar: Añadir el nodo al final y luego aplicar la operación flotar.
- Primero: Lleva asociado un coste constante O(1).
- Obtener Cima y Borrarla: Colocar el último elemento en la cima y hundir.

La inserción y borrado de elementos es muy eficiente. Los costes de las funciones son:

Función	Lista Enlazada	Montículo	Montículo Binomial	Montículo Fibonacci
insertar	O(1)	O(log n)	O(log n)	O(1)
mínimo/máximo	O(n)	O(1)	O(log n)	O(1)
borrar	O(n)	O(log n)	O(log n)	O(log n)

La función que crea un montículo a partir de una colección de valores, si se utiliza la función *Hundir*, puede llegar a tener un coste lineal.

Ordenación basada en Montículos: Algoritmo Heapsort

Con un montículo disponemos de una estructura de datos en la que encontrar el mínimo o el máximo es una operación de coste constante. Una vez extraído el primer elemento, restaurar la propiedad de montículo tiene un coste O(log n).

Sin pérdida de generalidad, asumimos que vamos a usar montículos de máximos. El funcionamiento del algoritmo heapsort es el siguiente:

- Se convierte el vector en un montículo.
- Se selecciona el máximo (la cima del montículo) y se incorpora al vector solución S. El montículo pierde un elemento y el vector S lo incorpora.
- Se restaura la propiedad del montículo sobre los elementos que restan. Así estamos en condiciones de volver a seleccionar el máximo.
- Al finalizar, el vector S contiene el vector de entrada ordenado de mayor a menor.

El coste se basa en la creación y restauración de la propiedad de montículo en cada extracción de la cima (primer elemento) del montículo. La creación es de coste lineal como hemos visto antes.

En cada paso del bucle se extrae el primer elemento y se restaura la propiedad de montículo con un coste total O(log n). Como realizamos n operaciones de coste O(log n), el algoritmo tiene un coste global O(n log n).

Grafos

Colección de nodos o vértices unidos por líneas o aristas. Se puede definir como la pareja G = (N, A), donde N es un conjunto finito de vértices o nodos, y A es un conjunto finito de líneas o aristas. Cada línea o arista es un par de la forma $A \in NxN$.

Los grafos permiten modelar problemas en los que existe una relación relevante entre los objetos que intervienen. Los nodos o vértices representarían los objetos y las aristas las relaciones entre ellos.

Si un grafo tiene pocas aristas, las listas de adyacencias resultan una estructura más costosa en espacio. Sin embargo, cuando el nº de nodos es cercano a n^2, el coste en espacio tiene el mismo coste.

La matriz de adyacencia es más sencilla de tratar, ya que no utilizan punteros.

Un ciclo, al cual también se le denomina circuito, es un camino simple que empieza y termina en el mismo vértice o nodo.

El grado de un vértice de un grafo no dirigido, es el número de aristas que salen o entran en él.

Un camino en un grafo dirigido es una secuencia finita de arcos entre dos vértices, tal que el vértice del extremo final de cada arco coincide con el del extremo inicial del arco siguiente.

La longitud de un camino es el número de aristas que contiene el grafo, pero no el número de nodos.

Tipos de Grafos

Tipo	Definición
Nulo	Es un Grafo sin vértices
Acíclico	Es un Grafo que no contiene ciclos
Etiquetado o Valorado	Es un Grafo en el que sus vértices tienen asociada información (etiqueta) que puede ser un nombre o valor, y, según el contexto, representa ponderación, peso, etc.
Simple	Es un Grafo en el que entre cada par de vértices existe, a lo sumo, una arista.
Multigrafo	Es un Grafo en el que entre cada par de vértices existe más de una arista.
Dirigido	Es un grafo al cual así se le llama si las líneas o aristas en A de dicho grafo están orientadas (indican un sentido), denominándose a estas flechas o arcos. En un grafo de este tipo con *n* vértices tiene un nº máximo de aristas de: $n * (n - 1)$
No Dirigido	Es un grafo definido así en el caso en que si las líneas o aristas en A de dicho grafo no se encuentran orientadas (no indican un sentido).
Completo	Grafo no dirigido con *n* vértices en el que el máximo nº de aristas es: $n * (n – 1)/2$
Conexo	Es un Grafo en el que para cualquier par de vértices existe un camino que los contiene.
Biconexo	Es un Grafo que no tiene puntos de articulación.
Fuertemente Conexo	Es un Grafo en el que si, para cada par de vértices distintos n y m, hay un camino de n a m y también hay un camino de m a n.
Árbol Libre	Es un Grafo Acíclico, Conexo y no dirigido. No tienen raíz y los nodos hijos no se encuentran ordenados.

Funciones de Manipulación de Grafos

Operación	Matriz de Adyacencia	Lista de Adyacencia
CrearGrafo	O(1)	O(1)
AñadirArista	O(1)	O(1)
BorrarArista	O(1)	O(n)
Etiqueta	O(1)	O(n)
AñadirVertice	O(n)	O(1)
BorrarVertice	O(n)	O(n+a)
EsAdyacente	O(1)	O(n)
Adyacentes	O(n)	O(1)

Recorrido de Grafos

Para la resolución de problemas que se pueden formular en forma de grafos, existen los distintos tipos de recorridos aplicables tanto a grafos dirigidos como no dirigidos:

- **Recorrido en Profundidad:** también conocido como *búsqueda primero en profundidad*. Inicialmente se marcan todos los nodos como no visitados y se selecciona un nodo *u* como punto de partida.
  A continuación, se marca como visitado y se accede a un nodo no visitado *v* adyacente al nodo *u*. Se procede recursivamente con el nodo *v*.

Al volver de la llamada o llamadas recursivas, si hay algún nodo adyacente que no ha sido visitado, se toma este como punto de partida, y se vuelve a ejecutar el procedimiento recursivo.

El recorrido termina cuando todos los nodos están marcados como visitados. Este recorrido sería equivalente al *recorrido en preorden de un árbol*.

El coste del recorrido en profundidad, si consideramos que *n* es el número de nodos y *a* es el número de aristas de un grafo, en su recorrido en profundidad sólo se hace una llamada a RecProfundidadRecursivo por cada nodo no visitado y se ejecuta *n* veces.

La obtención de los adyacentes depende de la forma de representación del grafo:

- Matriz de Adyacencia: los nodos adyacentes a otro nodo se obtienen recorriendo una fila de la matriz n x n, siendo el coste $O(n^2)$.
- Listas de Adyacencia: se recorre la lista de nodos adyacentes de un nodo para obtener sus nodos adyacentes. Como la suma de las longitudes de las listas de adyacencia que representan un grafo es *a*, el coste temporal sería O(n + a).

- **Recorrido en Amplitud o en Anchura:** También conocido como *búsqueda primero en anchura*. Inicialmente se marcan todos los nodos como no visitados y se selecciona un nodo *u* como punto de partida.

  A continuación, se marca como visitado y se visitan todos los nodos adyacentes a *u* que no hayan sido visitados, procediéndose entonces de forma similar con los adyacentes a cada uno de los nodos recién visitados.

  Se puede considerar como un *recorrido por niveles*. Primero se accede a los nodos que están a una arista de distancia del nodo inicial del recorrido, después a los que están a dos aristas de distancia, y así sucesivamente hasta que se visitan todos los nodos accesibles desde el inicial.

  El recorrido en anchura se emplea cuando hay que realizar una exploración parcial de un grafo infinito, o potencialmente muy grande, y también para calcular el camino más corto desde un punto del grafo hasta otro.

Punto de Articulación

Es un nodo *u* en un Grafo Conexo el cual, si se elimina junto a todas las aristas que inciden en él, dicho Grafo deja de ser Conexo.

TEMA 3 – ALGORITMOS VORACES

Algoritmos Voraces

Se aplican a problemas de optimización en los que la solución se puede construir paso a paso sin necesidad de reconsiderar decisiones ya tomadas, los cuales resuelven, genéricamente, el problema de *encontrar un conjunto de candidatos que constituyan una solución y que optimice una solución objetivo*.

Principalmente son utilizados en problemas de planificación de tareas y que se pueden modelar con grafos, en los que hay que realizar una búsqueda, cálculo de recorridos u optimización de pesos, entre otros.

Los problemas constan de *n* candidatos, de los cuales se pretende hallar un subconjunto o secuencia ordenada de los mismos, de manera que se optimice (maximice o minimice) una función objetivo.

Este esquema trabaja por etapas, considerando la elección de un candidato en cada etapa, teniendo que seleccionar en cada una de ellas el candidato más prometedor de los aún disponibles, y decidir entonces si se incluye o no en la solución.

Las características y elementos que pueden intervenir en este esquema son:

- Resolución de un problema de forma óptima.
- Conjunto inicial de candidatos (elementos que hay que planificar, vértices o aristas de un grafo, etc.).
- Conjunto de candidatos que ya han sido considerados y seleccionados (Inicialmente este conjunto está vacío).
- Conjunto de candidatos que ya han sido considerados y han sido rechazados (no volverán a ser considerados).
- Función que determina si un conjunto de candidatos es una solución al problema.
- Función Factible: determina si un conjunto es completable o factible, es decir, si añadiendo nuevos candidatos se puede alcanzar una solución que cumpla las restricciones del problema.
- Función de Selección: escoge al candidato más prometedor de los que todavía no se han considerado.
- Función Objetivo: representa el coste o valor de una solución (tiempo de proceso, longitud del camino, etc.) y es la que se quiere optimizar (maximizar o minimizar). Esta función no tiene porqué aparecer explícitamente en el algoritmo.

El esquema genérico es el siguiente:

```
fun Voraz(c: conjuntoCandidatos): conjuntoCandidatos
    sol ← ∅
    mientras c ≠ ∅ ∧ ¬ solucion(sol) hacer
        x ← seleccionar(c)
        c ← c \{x}
        si factible(sol ∪ {x}) entonces
            sol ← sol ∪ {x}
        fsi
    fmientras
    si solucion(sol) entonces devolver sol
        sino imprimir('no hay solución')
    fsi
ffun
```

En el esquema general aparecen los conjuntos y funciones antes mencionados:

- c: conjunto de candidatos.
- sol: conjunto de candidatos seleccionados.
- solucion(): función solución.
- seleccionar(): función de selección.
- factible(): función factible .

Estos conjuntos y funciones se tendrán que particularizar para cada problema concreto, siendo la descripción por pasos del algoritmo:

1. El conjunto inicial de candidatos ya escogidos está vacío.
2. La función de selección elige el mejor candidato de los aún no escogidos.
3. Si el conjunto resultante no es completable o factible, se rechaza el candidato y no se vuelve a considerar.
4. Si el conjunto resultante es completable, se incorpora el candidato al conjunto de candidatos escogidos.
5. Se comprueba si el conjunto resultante es una solución al problema y si quedan más candidatos.
6. En el caso de que no sea una solución y queden más candidatos que examinar se vuelve al paso 2.

La diferencia de este esquema con otros, es que nunca deshace una decisión ya tomada. Cuando se incorpora un candidato a la solución permanece hasta el final, y cuando se rechaza un candidato no se vuelve a tener en cuenta.

Si se encuentra un contraejemplo en el que el algoritmo no alcanza la solución óptima, podemos afirmar que dicho algoritmo diseñado no es correcto.

Se necesita una demostración de optimalidad para poder asegurar que el algoritmo alcanza la solución óptima.

Problema de la Mochila (Algoritmos Voraces)

El problema puede resolverse con algoritmos voraces, admitiendo tanto objetos enteros como fraccionables, pero siendo necesario que al menos siempre sean fraccionables.

Además, el esquema más adecuado y eficiente es el Algoritmo Voraz si se utiliza como criterio de selección escoger el objeto cuyo valor por unidad de peso sea el mayor de los que quedan.

Devolución de cambio de moneda (Algoritmos Voraces)

El problema se puede resolver con Algoritmos Voraces en los siguientes casos:

- Cuando se utiliza un número mínimo de monedas y la disponibilidad de cada tipo de moneda es ilimitada.
- Pago de una cantidad C > 0 utilizando un número mínimo de monedas, suponiendo como ilimitada la disponibilidad de cada tipo de moneda.

Esto aplica a cualquier sistema monetario, siendo indiferente si en él existe o no un tipo de moneda de 1.

La forma más típica de resolución de este problema con Algoritmos Voraces es en la que se dispone de *n* tipos de moneda T = {m^0, m^1, m^2, …, m^{n-1}}, siendo m > 1 y n > 0.

ARM: Árboles de Recubrimiento Mínimo

- **Algoritmo de Prim**

  Este algoritmo selecciona arbitrariamente un nodo del grafo como raíz del árbol ARM. En cada paso, se añade al árbol una arista de coste mínimo (u, v) tal que ARM U { (u, v) } sea también un árbol.

  El algoritmo continúa hasta que el ARM contiene n - 1 aristas. La arista (u, v) seleccionada en cada paso es tal que, o bien *u*, o bien *v*, está en el ARM.

Si el grafo se implementa mediante listas de adyacencia, y se utiliza además un montículo para representar los nodos candidatos pendientes, el coste es de O(n log n + a log n) = O(a log n).

Usar una **Lista de Adyacencia** resulta más apropiado que usar una Matriz de Adyacencia solo en el caso del **grafo disperso**.

En el caso de ser un grafo denso, el coste sería de $O(n^2 \log n)$, el cual es peor que el coste $O(n^2)$.

- **Grafo Disperso:** Es un grafo en el que su número de aristas es cercano a n.
- **Grafo Denso:** Grafo en el que su número de aristas es cercano a n^2.

- **<u>Algoritmo de Kruskal</u>**

  Partiendo del conjunto de aristas del ARM vacío, ordenadas de menor a mayor coste, en cada paso se selecciona la arista más corta o con menor peso que todavía no haya sido añadida o rechazada del ARM.

  Inicialmente, el ARM está vacío, y cada nodo de G forma una componente conexa. En los pasos intermedios, el grafo parcial formado por los nodos de G y las aristas del ARM consta de varias componentes conexas.

  Al final del algoritmo sólo queda una componente conexa, que es el ARM de G.

  Cuando en cada paso se selecciona la arista más corta que todavía no ha sido evaluada, se determina si une dos nodos pertenecientes a dos componentes conexas distintas.

  En caso afirmativo, la arista se añade al ARM, por lo que las dos componentes conexas ahora forman una. En caso contrario, esa arista se rechaza ya que forma un ciclo al unir dos nodos de la misma componente conexa.

  La demostración de optimalidad del algoritmo de Kruskal se realiza por inducción. Se trata de demostrar de que si el ARM es prometedor seguirá siéndolo en cualquier fase del algoritmo cuando se le añada una arista.

  El coste del algoritmo está en:

  - Ordenación de las aristas: O(a log a) → Equivale a O(a log n) y este a $O(a \log n^2)$.
  - Iniciar *n* conjuntos disjuntos: O(n).
  - Coste total del Algoritmo (se consideran ordenación e inicialización): O(a log n).

Comparando el algoritmo de Prim y el de Kruskal, uno u otro serán preferibles en los casos:

- <u>Grafo Denso:</u> Algoritmo de Prim con coste $O(n^2)$, con nº aristas cercanas a n^2.
- <u>Grafo Disperso:</u> Algoritmo de Kruskal con coste O(a log n), con nº aristas cercanas a n.

<u>Algoritmo Dijkstra</u>

Este algoritmo sigue el esquema Voraz. Con él se puede calcular el camino de coste o peso mínimo entre el origen y un único nodo del grafo. Para ello habría que detener el algoritmo una vez que el camino hasta dicho nodo ya se ha calculado.

Utiliza la noción de camino especial, el cual lo es si todos los nodos intermedios del camino desde un nodo origen a otro, estos pertenecen a S, es decir, si se conoce el camino mínimo desde el origen a cada uno de ellos.

Hará falta un array especial[] que en cada paso del algoritmo contendrá la longitud del camino especial más corto (si el nodo está en S), o el camino más corto conocido (si el nodo está en C) que va desde el origen hasta cada nodo del grafo.

Cuando se va a añadir un nodo a S, el camino especial más corto hasta ese nodo es también el más corto de todos los caminos posibles hasta él. Cuando finaliza el algoritmo todos los nodos están en S y, por lo tanto, todos los caminos desde el origen son caminos especiales.

Las longitudes o distancias mínimas que se esperan calcular con el algoritmo estarán almacenadas en el array especial[].

El coste del algoritmo Dijkstra será:

- Coste General (tiempo de ejecución): $O(n^2)$
- Tareas de Inicialización: O(n)
- Instrucción que elimina la raíz del montículo: O(log n)
- Coste cuando se actualiza el montículo con w ubicándolo según su valor de especial[w] cuando se encuentra un nodo *w* tal que, a través de *v*, se encuentra un camino menos costoso: O(log n)
- La raíz del montículo se elimina n-2 veces, y se realizan operaciones de flotar un máximo de a veces: O((n + a) log n)
- Implementación con montículo en los siguientes casos: O(a log n)
    - Grafo Conexo: $n - 1 <= a <= n^2$
    - Grafo Disperso: El número de aristas es pequeño y cercano a n
- Grafo Denso: $O(n^2 \log n)$, siendo preferible la implementación con montículo.

Camino de Coste Mínimo

Sea G = (N,A) un grafo dirigido, con pesos mayores o iguales que cero en sus aristas, en el que todos sus nodos son accesibles desde uno concreto considerado como nodo origen.

El algoritmo de Dijkstra determina la longitud del camino de coste, peso o distancia mínima que va desde el nodo origen a cada uno de los demás nodos del grafo.

También se puede utilizar para calcular el camino de coste o peso mínimo entre el origen y un único nodo del grafo. Para ello habría que detener el algoritmo una vez que el camino hasta dicho nodo ya se ha calculado. Su coste es:

- Coste General: $O(n^2)$
- Aplicado a la búsqueda del camino más corto entre cada par de nodos del grafo: $O(n^3)$

TEMA 4 - DIVIDE Y VENCERÁS

Divide y Vencerás

Técnica algorítmica basada en la descomposición de un problema en subproblemas de su mismo tipo, lo que permite disminuir la complejidad y, en algunos casos, paralelizar la resolución de los mismos, siendo su estrategia:

- Descomposición del problema en subproblemas de su mismo tipo o naturaleza
  Aquí se determina el número y tamaño de los subproblemas. Es uno de los pasos determinantes para la complejidad posterior del algoritmo.

- Resolución recursiva de los subproblemas
  Se realizan llamadas recursivas al algoritmo para cada uno de los subproblemas.
- Combinación, si procede, de las soluciones de los subproblemas
  Si es necesario, se combinan las soluciones parciales y se obtiene la solución del problema.

El algoritmo aplica el principio de inducción sobre los diversos ejemplares del problema, suponiendo con ello solucionados los subproblemas, y utiliza las soluciones parciales de los mismos para componer la solución al problema.

Para casos suficientemente pequeños, el esquema proporciona una solución trivial no recursiva (equivalente a la base de la inducción), la cual no requiere de nuevas descomposiciones.

El esquema general de este algoritmo es el siguiente:

```
fun DyV(problema)
    si trivial(problema) entonces
        dev solución-trivial
    sino hacer
        {p_1, p_2, ...p_k} ← descomponer(problema)
        para i ∈ (1..k) hacer
            s_i ← DyV(p_i)
        fpara
    fsi
    dev combinar(s_1, s_2, ...s_k)
ffun
```

El coste de este tipo de algoritmos depende de cuántas descomposiciones se realicen y del tipo de decrecimiento de las sucesivas llamadas recursivas a los subproblemas. En general se plantea una ecuación de recurrencia de decrecimiento del tamaño del problema en:

- Progresión Geométrica: $T(n) = aT(n/b) + cn^k$
- Progresión Aritmética: $T(n) = aT(n - b) + cn^k$

En ambos casos *a* es el número de subproblemas en los que se descompone el problema inicial, *b* es el factor de reducción del tamaño, y la expresión polinómica, indica el coste de la función de combinación de las soluciones parciales. Si k = 0 hablamos de un problema de reducción.

Ordenación Rápida (Quicksort)

Este algoritmo realiza el siguiente proceso:

- Toma un elemento cualquiera del vector denominado *pivote*.
- Toma los valores del vector que son menores que el pivote y forma un subvector. Se procede análogamente con los valores mayores o iguales.
- Se invoca recursivamente al algoritmo para cada subvector.

Para realizar la descomposición, se suele elegir como pivote el primer elemento del vector. Para almacenar los subvectores, se puede usar el propio vector situando los menores que el pivote a su izquierda, y los mayores a su derecha.

Una forma eficiente de realizar este proceso es comenzando a recorrer el vector por ambos extremos.

En el recorrido de izquierda a derecha, nos detenemos cuando se encuentre un elemento mayor que el pivote, y en el de derecha izquierda, nos detenemos al encontrar un elemento menor o igual que el pivote.

Entonces se intercambian ambos elementos, y se prosigue el recorrido. Cuando los recorridos se cruzan, el vector está descompuesto satisfactoriamente.

La descomposición en subproblemas tiene que ser equilibrada. En el caso peor en el que el pivote fuera el menor (o el mayor) valor del vector generaríamos un vector de un solo elemento y otro vector con los demás.

El algoritmo necesita O(n) llamadas recursivas. Como la función *Pivotar* tiene un coste lineal, el algoritmo tiene un coste de $O(n^2)$.

Si se utiliza como pivote la mediana del vector, o si requiere un tiempo lineal, se puede pivotar alrededor de ella en tiempo O(n) y, entonces, el algoritmo quicksort mejorado tendría un coste de O(n log n), incluso en el caso peor.

Sin embargo, la constante oculta se vuelve tan grande que lo hace menos eficiente que otros algoritmos, como la ordenación con montículo.

Ordenación por Fusión (Mergesort)

Dado un vector de enteros, se plantea dividir este por la mitad e invocar al algoritmo para ordenar cada mitad por separado. Tras dicha operación, solo queda fusionar esos dos subvectores ordenados en uno sólo, también ordenado.

La fusión (merge) se hace tomando sucesivamente el menor elemento de entre los que queden en cada uno de los dos subvectores.

Se utiliza el algoritmo de ordenación por inserción para los tamaños pequeños. La combinación de las soluciones parciales es la mezcla o fusión de los subvectores ordenados.

Puesto que tanto la separación en dos subvectores como su posterior fusión tiene coste lineal, el coste del algoritmo es O(n log n).

Cálculo del elemento mayoritario en un vector

Dado un vector v[1..n] de números naturales, se quiere averiguar si existe un elemento mayoritario, es decir, que aparezca al menos *n/2 + 1* veces en el vector.

Siguiendo los pasos indicados en la exposición del esquema, la descomposición puede ser en 2 subproblemas de tamaño n/2, que es la más eficiente.

En cuanto a la combinación de los subproblemas resueltos, el algoritmo nos debe proporcionar, bien el valor mayoritario, o bien indicarnos que éste no existe. La función de recurrencia asociada es la siguiente: $T(n) = 2T(n/2) + cn$

En el peor caso, es necesario recorrer el vector tras las llamadas recursivas para contar los elementos mayoritarios. En este caso tenemos $a = b = 2$ y $k = 1$ y, por tanto: $T(n) \in O(n \log n)$

Búsqueda Binaria en un Vector ordenado

Se dispone de un vector V[1…n] que almacena números enteros en orden estrictamente creciente. Si se quiere averiguar si existe algún elemento que cumpla V[i] = i, el algoritmo que mejor resuelve este problema es el de Divide y Vencerás con coste O(log n).

Liga de Equipos

Supongamos que *n* equipos (con $n = 2^k$) desean realizar una liga. Las condiciones en las que se celebra el torneo son las siguientes: cada equipo puede jugar un partido al día, y la liga debe celebrarse en n - 1 días.

El problema plantea la realización de un calendario en el que por cada par de equipos se nos indique el día en que juega. La solución tendrá por tanto forma de matriz simétrica con $M[e_i,e_j] = d_{ij}$, que indica que el equipo e_i juega el día d_{ij} con el equipo e_j.

El resultado del problema se puede almacenar en una tabla T[1..n, 1..n] simétrica con $T[i, j] = d_{ij}$, siendo d_{ij} el día del partido entre el equipo e_i y el equipo e_j.

El algoritmo por tanto necesita tener como argumento, tanto los equipos que juegan, como el día en que empieza el torneo.

El coste del algoritmo de combinación es cuadrático, considerando n = j - i + 1 el tamaño del problema. En tal caso, la ecuación de recurrencia queda $T(n) = 2T(n/ 2) + cn^2$ con a = 2, b = 2 y k = 2, con lo que siendo $a < b^k$, la ecuación tiene solución cuadrática, con coste del algoritmo $O(n^2)$.

TEMA 5 - PROGRAMACIÓN DINÁMICA

Programación Dinámica

Algoritmo mediante el cual se reduce el coste de ejecución de un algoritmo mediante la memorización de soluciones parciales empleadas para llegar a la solución final, siendo un algoritmo aplicable también en problemas que pueden resolverse con *Divide y Vencerás* (no obstante, cuando ocurren las repeticiones, es más conveniente recurrir al esquema de Programación Dinámica, el cual reutiliza los resultados ya calculados), así como a muchos problemas de optimización cuando se cumple el ***Principio de Optimalidad de Bellman (****Dada una secuencia óptima de decisiones, toda subsecuencia es, a su vez, también óptima****)***.

El problema se divide en subproblemas de tamaño menor al original, planteándose dichas divisiones en forma de algoritmos recursivos. Para que el algoritmo sea eficiente, éste debe cumplir que no contenga llamadas repetidas en la secuencia de llamadas recursivas.

La resolución del problema requerirá buscar la solución óptima para muchas subsecuencias que pudieran formar parte de la secuencia óptima final.

Los resultados parciales se almacenan en una tabla, de la cual se toman una vez que el algoritmo los vuelve a necesitar. Los primeros datos que se almacenan corresponden a los subproblemas más sencillos, a partir de los que se van construyendo de forma incremental las soluciones a subproblemas mayores, hasta llegar a la solución del problema completo.

La reducción del coste en tiempo supone un incremento en el coste espacial, ya que el almacenamiento de las soluciones parciales requiere incrementar el espacio dedicado a los datos. La forma en que se aplica el algoritmo es muy dependiente del problema y de las estructuras de datos que se utilicen en su resolución.

Multiplicación Asociativa de Matrices

Consiste en calcular la matriz producto M de N matrices. El producto de una matriz A por otra B de dimensiones (m x n) y (n x p), respectivamente, es una matriz C de dimensión (m x p).

Se requieren (m x n x p) productos escalares para realizar la multiplicación de dos matrices.

El objetivo del problema consiste en buscar un algoritmo que halle el número mínimo de multiplicaciones escalares con el que se puede realizar el producto de una secuencia de matrices.

En el algoritmo del problema existen dos bucles anidados de coste O(n), en los que se llama a la función *MinMultiple*, también de coste O(n), siendo los costes del mismo:

- Temporal: $O(n^3)$
- Espacial: $O(n^2)$, debido a que se construye una tabla NxN.

Distancia de Edición entre dos Cadenas

Se consideran dos cadenas de caracteres, X e Y, de un alfabeto finito. La cadena $X = X_1, X_2, \cdots, X_n$, tiene longitud n, y la cadena $Y= Y_1, Y_2, \cdots, Y_m$, tiene longitud m. La cadena X se puede transformar en la cadena Y realizando los siguientes tipos de cambios:

- Borrar un carácter de X
- Insertar uno de los caracteres de Y en X
- Sustituir un carácter de X por uno de los de Y

El objetivo de este problema es encontrar el número mínimo de cambios para transformar la cadena X en la cadena Y.

El coste temporal, que es el mismo que el espacial, O(nm), puede venir dado por:

- Los 2 bucles anidados usados para construir la tabla
- El tamaño de la tabla

Emisión de Sellos de un País

Un país emite n sellos diferentes, de valores naturales positivos $s_1, s_2, \ldots, s_n$.

Si se quiere enviar una carta, sabiendo que la correspondiente tarifa postal es T, el esquema de Programación Dinámica es el más adecuado para resolver este problema cuando se quiere saber de cuantas formas diferentes podemos franquear exactamente la carta (no importando el orden de los sellos).

El coste del algoritmo es:

- Temporal: O(nT)
- Espacial: O(T)

Problema de la Mochila

Este problema puede resolverse con el algoritmo Programación Dinámica, admitiendo únicamente objetos enteros. El coste con el que lo resuelve es:

- Sin Optimización de memoria: O(nm)
- Con Optimización de memoria: O(m)

Devolución de cambio de moneda

El problema se puede resolver con Programación Dinámica en los siguientes casos:

- Cuando se utiliza un número mínimo de monedas y la disponibilidad de cada tipo de moneda es ilimitada (Coste: O(nC)).
- Para una cantidad C > 0, no cumpliéndose T = {m^0, m^1, m^2, …, m^n}, siendo m>1 y n>0.

Esto aplica a cualquier sistema monetario, siempre que disponga de un tipo de moneda de 1.

TEMA 6 - VUELTA ATRÁS

Vuelta Atrás

Este esquema realiza un recorrido en profundidad del grafo implícito de un problema, podando aquellas ramas para las que el algoritmo puede comprobar que no pueden alcanzar una solución al problema.

Las soluciones se construyen de forma incremental. A un nodo del nivel k del árbol le corresponderá una parte de la solución (solución o secuencia k-prometedora) construida en los k pasos dados en el grafo para llegar a dicho nodo.

Si la solución parcial construida en un nodo no se puede extender o completar, este será un nodo de fallo, retrocediendo entonces el algoritmo hasta un nodo en el que quedan ramas pendientes de ser exploradas.

En su forma más básica, el algoritmo de Vuelta Atrás se asemeja a un recorrido en profundidad dentro de un grafo dirigido, siendo este un árbol o un grafo sin ciclos.

Se van construyendo las soluciones parciales a medida que se avanza en el recorrido por el grafo implícito.

Un recorrido tiene éxito si se llega a completar una solución, no teniendo éxito si ésta no se puede seguir completando en alguna etapa, en cuyo caso el recorrido vuelve atrás (como en un recorrido en profundidad), eliminando los elementos que se han añadido en cada fase.

El recorrido de búsqueda de solución continúa a partir de un nodo al que se llega, el cual tiene algún nodo vecino sin explorar.

El objetivo del algoritmo es encontrar una o todas las soluciones posibles al problema. Si encuentra una, se detiene al encontrarla, evitando así la construcción del resto del grafo del problema, lo cual supone una gran ganancia del algoritmo en eficiencia.

No obstante, cuando para un mismo problema pueden ser aplicables tanto un Algoritmo Voraz como el Algoritmo de Vuelta Atrás, será siempre más eficiente el primero de ellos.

El esquema general de Vuelta Atrás es el siguiente:

```
fun VueltaAtras (v: Secuencia, k: entero)
    IniciarExploraciónNivel(k)
    mientras OpcionesPendientes(k) hacer
        extender v con siguiente opción
        si SoluciónCompleta(v) entonces
            ProcesarSolución(v)
        sino
            si Completable (v) entonces
                VueltaAtras(v, k+1)
            fsi
        fsi
    fmientras
ffun
```

Problema de la Mochila

Este problema puede resolverse con el algoritmo Vuelta Atrás, admitiendo únicamente objetos enteros. El coste con el que lo resuelve es $O(2^n)$.

Coloreado de Grafos

En este problema se desea asignar un color de un conjunto de *m* colores a cada vértice de un grafo conexo, de manera que no haya dos vértices adyacentes que tengan el mismo color.

La convención de usar colores tiene su origen en el problema de colorear mapas en un plano. Este problema se transforma en el coloreado de mapas, asignando un vértice a cada país y un enlace entre los vértices que representan a países vecinos.

Los datos de entrada al algoritmo son el grafo, el número de colores disponibles, y la posición del vértice al que toca asignar color. Los datos de salida son la solución (que también es de entrada) y un indicador de que se ha conseguido con éxito la asignación de colores. Cuando se procesa el último nodo del grafo (k = n), el algoritmo termina con éxito.

Mientras eso no ocurre se van asignado colores al siguiente nodo del grafo, que está en la posición k + 1 del vector solución. Si un color no es válido se pasa a probar el siguiente, incrementando el valor asignado a v [k].

El espacio de búsqueda de este algoritmo tiene la forma del árbol mostrado anteriormente, por lo que una cota al coste del algoritmo es $O(n \cdot m^n)$, siendo el número de nodos del árbol por el coste *n* de invocación a la función *Completable* para cada uno. El coste habitual es $O(m^n)$.

Reparto Equitativo de Activos

Cada uno de los *n* activos de una sociedad que se va a disolver, en la que hay que repartirlos entre dos socios, estos tienen un valor entero positivo.

Los socios se quieren repartir dichos activos a medias, y quieren conocer todas las posibles formas que tienen de dividirlos en dos subconjuntos disjuntos, de forma que cada uno de ellos tenga el mismo valor entero.

Este algoritmo comprueba si se ha alcanzado una solución, lo cual se cumple si se han asignado todos los activos (k = n) de forma que la suma asignada a cada socio sea la misma.

Si aún quedan activos pendientes se construyen los vectores k + 1-prometedores que corresponden a asignar el siguiente activo k + 1 al socio 1 o al socio 2.

En cada caso se comprueba si la asignación es válida mediante una llamada a la función *completable*, con la cual la asignación del activo *k* a un determinado socio no lleva a que los valores asignados a ese socio superen la mitad del total.

La llamada a la función que reparte los activos requiere un procesamiento previo para calcular la suma total de los activos a dividir y en inicializar las variables de llamada. Antes de la llamada se comprueba que el valor total sea par, para que, de este modo, sea posible el reparto en cantidades enteras.

El coste viene dado por el número máximo de nodos del espacio de búsqueda. Como cada nodo del árbol se divide en dos, que corresponden a asignar el siguiente activo al socio 1 o al socio 2, y, como la profundidad del árbol viene dada por el número de activos *n*, entonces el coste está acotado por $O(2^n)$.

Subconjuntos de suma dada

Se dispone de un conjunto A de *n* números enteros sin repeticiones (tanto positivos como negativos) almacenados en una lista.

Dados dos valores enteros m y C, siendo m < n, se desea resolver el problema de encontrar todos los subconjuntos de A compuestos por exactamente *m* elementos tal que, la suma de los valores de esos *m* elementos, sea C.

Tenemos que recorrer un árbol de soluciones candidatas, pero siguiendo aquellas ramas que tengan opción de convertirse en una solución (k-prometedoras).

Dependiendo de la representación del vector solución y de la estrategia de ramificación (generación de hijos), podríamos considerar distintas aproximaciones.

La aproximación más adecuada de ellas consiste en representar la solución como un vector de *n* booleanos, con lo que los descendientes de un nodo de nivel k serían las opciones de tomar o no el valor k + 1 del vector de entrada (tendríamos siempre dos opciones).

Cuando m tiende a n, el número de nodos sería 2^n, que es mejor que n!. El inconveniente es que con ello se trata de evitar el considerar permutaciones, es decir, tomar los mismos números, pero en orden diferente.

El conjunto A de números se representa por un vector de enteros. Las soluciones candidatas las representaremos por un vector de booleanos en el que una posición con valor cierto indica que el número correspondiente a esa posición se incluye en la solución.

Por cada nivel, el árbol se divide como máximo en dos ramas (si no se ha llegado a tener m posiciones con valor cierto), luego una cota superior, su coste, es $O(2^n)$.

TEMA 7 - RAMIFICACIÓN Y PODA

Ramificación y Poda

Esquema usado para explorar un grafo dirigido implícito que busca la solución óptima de un problema. Utiliza la función que se quiere optimizar para establecer preferencias del orden en que los nodos serán explorados.

Se requiere de una cola de prioridad para gestionar los nodos activos, ya que el recorrido se dirige por el nodo activo más prometedor.

Seleccionado un nodo, se procede con una fase de ramificación, en la que se generan distintas extensiones de la solución parcial correspondiente a dicho nodo.

Se podan las ramas tanto que no pueden llegar a una solución, como aquellas que no mejoran el valor asignado a la solución por la función que se quiere optimizar, calculando por cada nodo una cota optimista del posible valor de las soluciones que pueden construirse a partir del mismo.

Si la cota indica que estas soluciones serán con seguridad peores que una solución factible ya encontrada (o que una cota pesimista de las soluciones posibles), no tiene sentido seguir explorando esa parte del grafo y se poda. Se realiza una poda por factibilidad y otra por cota.

El esquema general de Ramificación y Poda es el siguiente:

```
fun RamificacionYPoda (nodoRaiz, mejorSolución: TNodo, cota: real)
    monticulo = CrearMonticuloVacio()
    cota = EstimacionPes(nodoRaiz)
    Insertar(nodoRaiz, monticulo)
    mientras ¬ MonticuloVacio?(monticulo) ∧
              EstimacionOpt(Primero(monticulo)) ≤ cota hacer
       nodo ← ObtenerCima(monticulo)
       para cada hijo extensión válida de nodo hacer
          si solución(hijo) entonces
             si coste(hijo) ≤ cota entonces
                cota ← coste(hijo)
                mejorSolucion ← hijo
             fsi
          sino
             si EstimacionOpt(hijo) ≤ cota entonces
                Insertar(hijo, monticulo)
                si EstimacionPes(hijo) < cota entonces
                   cota ← EstimacionPes(hijo)
                fsi
             fsi
          fsi
       fpara
    fmientras
ffun
```

El algoritmo acumula en un montículo los elementos que pueden ser soluciones al problema una vez estén completados.

En el montículo se mantienen ordenados por una estimación optimista (EstimaciónOpt) o a la baja (si consideramos una minimización) del valor que puede alcanzar.

Usa una cota para podar los nodos para los que sea posible estimar que, en el mejor de los casos, no van a poder dar lugar a una solución mejor que la asociada a la cota.

Si la estimación optimista de la cima del montículo, en un algoritmo de optimización, es mayor o igual que la cota superior hasta el momento, mejorando por ello tanto la solución existente hasta ese momento como la estimación optimista de la cima, diremos que el algoritmo, definitivamente, ha terminado (ha encontrado la solución).

El coste del problema dependerá del tamaño del árbol de posibles soluciones, por lo que, en función de dicho tamaño, daremos una cota superior al coste calculado en función del mismo.

Problema de la Mochila (Ramificación y Poda)

Este problema puede resolverse con el algoritmo Ramificación y Poda, admitiendo únicamente objetos enteros. El coste con el que lo resuelve es $O(2^n)$.

Se da además cuando el objeto puede meterse en la mochila, no meterse, o meter solo la mitad de este, obteniendo con ello la mitad del beneficio.

Devolución de cambio de moneda (Ramificación y Poda)

El problema se puede resolver con Ramificación y Poda cuando se utiliza un número mínimo de monedas y, la disponibilidad de cada tipo de moneda, es ilimitada.

Esto aplica a cualquier sistema monetario, siempre que disponga de un tipo de moneda de 1.