Algoritmos

Juan Dodyk

Índice

1 Algoritmos

1.1 Estructuras de datos

Asumimos que crear un arreglo toma tiempo lineal y que acceder toma tiempo constante.

1.1.1 Lista, vector, stack, queue

Una *lista* es un conjunto de nodos $u_1, \ldots, u_n$, los cuales guardan un elemento $f(u_i)$, un enlace al siguiente $s(u_i) = u_{i+1}$ (o nil) y uno al anterior $a(u_i) = u_{i-1}$ (o nil), y un puntero al primero y al último. La estructura se puede implementar fácilmente para que soporte insertar al principio, al final, borrar nodo, reverse (que manda $u_1, \ldots, u_n$ a $u_n, u_{n-1}, \ldots, u_1$) y merge (dadas dos listas $u_1, \ldots, u_n$ y $v_1, \ldots, v_m$ devuelve la lista $u_1, \ldots, u_n, v_1, \ldots, v_m$) en tiempo constante.

Un *vector* es un arreglo que soporta la operación **push-back**, que agrega al final del arreglo un elemento dado, en tiempo constante amortizado. La implementación de **push-back** es así: si el arreglo subyacente tiene tamaño suficiente para albergar un elemento más, se lo agrega; si no, se crea un arreglo del doble de longitud del arreglo actual, se copia el contenido del arreglo actual en el nuevo, se elimina el actual y se sigue sobre el nuevo. Se ve que la complejidad de esta operación es $O(1)$ amortizado.

Un *stack* es una estructura que soporta las operaciones **empty**, **top**, **push** y **pop** en tiempo constante. La primera informa si contiene algún elemento, la segunda devuelve el último insertado, la tercera inserta y la última quita el último. Se puede implementar sobre un arreglo si tenemos una cota superior útil de la cantidad de elementos que vamos a manejar o, si no, con un vector.

Una *queue* es una estructura que soporta las operaciones **empty**, **front**, **push** y **pop** en tiempo constante. La primera informa si contiene algún elemento, la segunda devuelve el primer elemeto insertado, la tercera inserta y la última quita el primer elemento insertado. Si tenemos una cota superior útil de la cantidad de operaciones **push** se puede implementar con un arreglo que contiene dos índices, uno qb al primer elemento y otro qf al último: **empty** es $qb = qf$, **front** es $arr[qb]$, **push** es $arr[qf] = e, qf = qf + 1$, **pop** es $qb = qb + 1$. Si sólo tenemos una cota superior del tamaño también podemos implementarla en un arreglo, con esas operaciones, sólo que $qf = qf + 1$ y $qb = qb + 1$ pasan a ser $qf = (qf + 1) \bmod tam$ y $qb = (qb + 1) \bmod tam$.

Último menor. Dada una secuencia de números $a_1, \ldots, a_n$ queremos saber, para todo $i \in [n]$, el índice $j = f(i)$ máximo con $j < i$ y $a_j \leqq a_i$, si es que hay.

El algoritmo obvio es, para todo i, mirar los $j = i - 1, i - 2, \ldots$ hasta que $a_j \leqq a_i$. Ahora si $a_j \geqq a_k$, con $j < k$, j nunca va a ser usado después de k. Entonces sólo tenemos que guardar la secuencia $j_1, \ldots, j_r < i$ con $a_{j_1} < \cdots < a_{j_r}$. Para actualizarla, tomamos a_i y lo vamos comparando contra $a_{j_r}, a_{j_{r-1}}, \ldots$. Cuando $a_{j_s} \leqq a_i$ sabemos que $f(i) = j_s$. Pero notemos que en la siguiente iteración la secuencia va a ser $j_1, \ldots, j_s, i$ (si $a_{j_s} < a_i$; si no borramos j_s también). Entonces la podemos mantener con estructura de pila. El tiempo es $O(n)$, ya que en total son $O(n)$ operaciones **top**, $O(n)$ **push** y $O(n)$ **pop**, y cada una se puede resolver en tiempo constante.

DFS. Dado un grafo G dirigido o no y un vértice u queremos saber a cuáles vértices se puede llegar desde u. El recorrido DFS consiste en avanzar sin pasar dos veces por el mismo vértice. Cuando no se puede avanzar más, se retrocede y se intenta de nuevo. Esto se puede implementar con una stack. Al principio marcamos u como visitado y lo ponemos en la stack. Luego iteramos, mientras la stack no sea vacía, tomando el último vértice, quitándolo de la stack y visitando y agregando a todos sus vecinos no visitados al final. Tarda tiempo lineal en el tamaño de la componente conexa de u. Notemos que del DFS se puede extraer un árbol con raíz u cuyos caminos bastan para conectar u con todos los vértices v tales que $u \rightsquigarrow v$.

Si G es dirigido, el DFS divide las aristas en cuatro clases: *tree edges*, las que quedan en el árbol, *forward edges*, las que conectan un vértice con un descendiente suyo, *back edges*, las que conectan un vértice con un ancestro suyo, y *cross edges*, las que conectan un vértice con uno de otra rama. Veamos una manera de determinar de qué tipo es cada arista. A cada vértice le asignamos un color; primero todos son pintados de blanco. Cuando visitamos un vértice por primera vez lo pintamos de gris y no lo eliminamos de la pila. Cuando volvamos a encontrarnos con ese vértice, lo pintamos de negro y lo sacamos de la pila. Notar que esto no afecta la complejidad. En cada iteración, si nos encontramos con un vértice blanco v, ponemos $pre[u] = k$ y luego $k = k + 1$, donde k va a ser un contador; si es gris, $pos[u] = k$. Ahora se ve que $vw \in E$ es tree o forward edge si $pre[v] < pre[w] < pos[w] < pos[v]$, es back edge si $pre[w] < pre[v] < pos[v] < pos[w]$ y es cross edge si $pre[w] < pos[w] < pre[v] < pos[v]$.

Topological sort. Dado un grafo G dirigido y sin ciclos con $V = [n]$ se quiere un orden $\pi \in S_n$ tal que si $ij \in E$ entonces $\pi_i^{-1} < \pi_j^{-1}$.

Largamos DFS desde cada vértice. Entonces un topological sort viene dado por tomar los vértices en el orden decreciente de *pos*. En efecto, como G no tiene ciclos no va a haber back edges así que si $vw \in E$, $pos[w] < pos[v]$.

Strongly connected components. Dado G dirigido sea la relación $\sim \subset V^2$ dada por $u \sim v$ si y sólo si $u \rightsquigarrow v$ y $v \rightsquigarrow u$. Es de equivalencia. Determinar las clases de equivalencia en tiempo lineal.

Armamos un grafo con vértices $V/\sim$: $[u], [v]$ es arista (con $u \not\sim v$) si $u \rightsquigarrow v$. Se ve que es acíclico. Si de $[u]$ no salen aristas, un DFS sobre u determina $[u]$. Supongamos que $uv \in E$. Veamos que un DFS sobre el grafo va a cumplir que $\max_{w \sim u} pos[w] > \max_{w \sim v} pos[w]$. En efecto, si $[u]$ se visita antes que $[v]$, el primero que visita de $[u]$ va a tener mayor *pos* que cualquiera de $[v]$; el otro caso es trivial. En particular, el vértice u con *pos* máximo cumple que a $[u]$ no entran aristas. Entonces si revertimos todas las aristas del grafo, el vértice u de mayor *pos* va a cumplir que de $[u]$ no van a salir aristas, así que hacemos DFS desde u y encontramos $[u]$; cuando eliminemos estos vértices, el vértice no eliminado con *pos* mayor va a generar otra componente, y así. En resumen, el algoritmo es: largamos DFS sobre el grafo con las aristas invertidas y luego largamos DFS sobre el grafo original sobre los vértices en orden decreciente de *pos*.

BFS. Si u, v son dos vértices en un grafo, la distancia $d(u, v)$ entre ellos es la longitud del camino más corto de u a v. Dado u calcular la distancia de u a todos los demás vértices en tiempo lineal.

Hacemos una búsqueda en ancho. Esto es, en cada paso tomamos un vértice v y miramos sus vecinos w que no fueron previamente visitados, poniendo $d(u, w) = d(u, v) + 1$; cuando terminamos de ver sus vecinos seguimos con los vecinos de sus vecinos. Esto se puede implementar usando la estructura de una queue. Primero marcamos u como visitado, ponemos $d(u, u) = 0$, y lo metemos en la queue. Luego, mientras la queue contenga elementos, pedimos v con **front**, miramos sus vecinos w no visitados, ponemos $d(u, w) = d(u, v) + 1$ y los agregamos a la queue.

1.1.2 Binary heaps y Fibonacci heaps

Una *heap* es un bosque con raíces (U, F) y una función $f : U \to \mathbb{R}$ de manera que si $uv \in F$ entonces $f(u) \leqq f(v)$. Veamos dos maneras de implementarlo: la primera es sobre un árbol binario y la segunda, sobre un bosque Fibonacci.

Una *binary heap* es un orden $u_1, u_2, \ldots, u_n$ de números tal que si $i = \lfloor \frac{j}{2} \rfloor$ entonces $f(u_i) \leqq f(u_j)$. La estructura soporta las operaciones **extract-min** y **decrease-key** en tiempo $O(\log n)$, y se puede armar, dados los números y la función f, en tiempo lineal. La primera elimina u_1. Para esto, ponemos $u_1 = u_n$, $n = n - 1$ y hacemos sift-down: ponemos $i = 1$; mientras hay $j \leqq n$ con $2i \leqq j \leqq 2i + 1$ y $f(u_j) < f(u_i)$, intercambiamos u_i y u_j y ponemos $i = j$. La segunda decrece el valor de $f(u_i)$ para algún i. Para eso, mientras $i > 1$ y $f(u_j) > f(u_i)$ para $j = \lfloor \frac{i}{2} \rfloor$, intercambiamos u_i y u_j y ponemos $i = j$. Se ve que las dos operaciones dan binary heaps y toman tiempo $O(\log n)$. Para construir la cola ponemos un orden cualquiera de $u_1, \ldots, u_n$ y vamos desde $i = \lfloor \frac{n}{2} \rfloor$ hasta 1 haciendo sift-down. El costo se ve que es $O(\log \frac{n}{i})$ así que en total es $O(n)$.

Una *Fibonacci heap* es un bosque (U, F) y un conjunto $T \subset U$ que cumple:

F1 Si $uv \in F$, $f(u) \leqq f(v)$.
F2 Si v es el i-ésimo hijo de u entonces tiene al menos $i - 2$ hijos; si $v \notin T$ entonces tiene al menos $i - 1$ hijos.
F3 Si u y v son dos raíces distintas entonces $\deg^+(u) \neq \deg^+(v)$.

Se ve que cada vértice v tiene al menos $F_{\deg^+(v)+1}$ descendientes (v incluído), donde F_i es el i-ésimo número de Fibonacci ($F_0 = F_1 = 1$, $F_{n+2} = F_{n+1} + F_n$) así que si hay r raíces entonces $|U| \geqq F_{r+2} - 2$, por lo que $r \leqq \lfloor 2 \log_2(|U| + 2) \rfloor + 1$. Describimos la Fibonacci heap con la siguiente estructura:

E1 Para cada $u \in U$, una lista con los hijos en orden; específicamente es una función $ult : U \to U \cup \{nil\}$, tal que $ult(u)$ es el último hijo de u (o nil si no tiene hijos), $ant : U \to U \cup \{nil\}$, tal que $ant(u)$ es el anterior hijo de u o nil si es el primero y $sig : U \to U \cup \{nil\}$, tal que $sig(u)$ es el siguiente hijo de u o nil si es el último.
E2 Una función $p : U \to U$, donde $p(u)$ es el padre de u o u si es raíz.
E3 La función $\deg^+ : U \to \mathbb{N}_0$.
E4 Una función $b : \{0, \dots, t\} \to U \cup \{nil\}$, donde $t = \lfloor 2 \log_2(|U|+2) \rfloor + 1$, tal que $b(d^+(u)) = u$ para cada raíz u.
E5 Una función $l : U \to \{0, 1\}$ tal que $l(u) = 1$ si y sólo si $u \in T$.

Veamos que si al principio $|U| = p$ y hacemos n operaciones **extract-min** y m **decrease-key**, la estructura puede ser mantenida en tiempo $O(m + p + n \log p)$.

Tenemos dos operaciones internas:

repair(r):
si $b(\deg^+(r)) = nil$, poner $b(\deg^+(r)) = r$, $p(r) = r$;
si no, sea $s = b(\deg^+(r))$;
si $f(r) \leqq f(s)$, poner $sig(ult(r)) = s$, $ant(s) = ult(r)$, $ult(r) = s$ (o sea hacer que s sea el último hijo de r), $p(s) = r$, $p(r) = r$, $b(\deg^+(r)) = nil$, $\deg^+(r) = \deg^+(r) + 1$ y **repair**(r);
si no, poner $sig(ult(s)) = r$, $ant(r) = ult(s)$, $ult(s) = r$ (o sea hacer que s sea el último hijo de r), $p(r) = s$, $b(\deg^+(s)) = nil$, $\deg^+(s) = \deg^+(s) + 1$ y **repair**(s).

make-root(u):
si $p(u) = v \neq u$:
{borrar la arista vu:
si $sig(u) \neq nil$, poner $ant(sig(u)) = ant(u)$;
si $ant(u) \neq nil$, poner $sig(ant(u)) = sig(u)$;
si $sig(u) = nil$, poner $ult(v) = ant(u)$;
poner $\deg^+(v) = \deg^+(v) - 1$;
si $p(v) = v$, poner $b(\deg^+(v) + 1) = nil$ y **repair**(v);
repair(u);
si $l(v) = 0$ (o sea $v \notin T$), poner $l(v) = 1$ (agregar a T);
si no, poner $l(v) = 0$ (sacarlo de T) y **make-root**(v)}.

Lo que hace **repair**(r) es hacer que se mantenga **F3** cuando queremos que r sea raíz. Ahora la operación **extract-min** lo que hace es buscar i en $\{0, \dots, t\}$ tal que $f(b(i))$ sea mínimo, poner $f(b(i)) = nil$ e ir aplicando **repair** a sus hijos. La operación **decrease-key** simplemente llama a **make-root** del vértice u tal que $f(u)$ decreció, si $f(u) < f(p(u))$.

Vamos a analizar la complejidad. Veamos que si en un momento dado se hicieron en total r **repair** y t **make-root** (sin contar las llamadas recursivas) entonces el tiempo fue $O(r + t)$. Cada llamada recursiva de **repair** borra una raíz, que la tuvo que agregar una llamada no recursiva, así que éstas suman $O(r)$. Cada llamada recursiva de **make-root** borra un elemento de T, que lo tuvo que agregar una llamada no recursiva, así que suman $O(t)$, y en total fueron a lo sumo $4t$ llamadas a **repair**. Así que todas las llamadas suman en total $O(r + t)$. Ahora **extract-min** llama $O(n \log p)$ veces a **repair** y **decrease-key** llama m veces a **make-root** así que el tiempo total es $O(m + n \log p)$.

Armar la Fibonacci heap se puede hacer en tiempo lineal: empezamos con $b = nil$, todos los nodos con $ant = sig = ult = nil$, $l = \deg^+ = 0$ y hacemos **repair** con cada uno. Esto termina de sumar $O(m + p + n \log p)$.

Dijkstra. Dado un grafo G, una función $w : E \to \mathbb{R}_{\geqq 0}$ y un vértice u, para todo $v \in V$ encontrar la longitud $d(v)$ del camino más corto de u a v.

La idea es calcular las distancias en orden de menor a mayor. Partimos de $d(u) = 0$, $d(v) = w(uv)$ para $uv \in E$, $d(v) = \infty$ para el resto. Supongamos que sabemos que las distancias de $U = \{u_1 = u, \ldots, u_k\}$ son $d(u_1) \leqq \ldots \leqq d(u_k)$, que si $v \notin V$ entonces $d(u_k) \leqq d(v)$ y la longitud del camino más corto de u a v pasando únicamente por los vértices de U es $d(v)$. Entonces sea $v \notin V$ con $d(v)$ mínimo. Veamos que $U \cup \{v\}$ cumple lo mismo. Primero supongamos que hay un camino p de u a v longitud $l < d(v)$. Sea $w \in p$ el primero tal que $w \notin U$ (hay uno porque v cumple). Entonces $d(w) \leqq l < d(v)$ (usamos que los pesos son no negativos), que contradice que $d(v)$ es mínimo. Entonces la distancia de s a v es precisamente $d(v)$. Ahora actualizamos $d(w)$ para los $w \notin U \cup \{v\}$: miramos las $vw \in E$ y ponemos $d(w) = \text{mín}\{d(w), d(v) + w(vw)\}$. Claramente se cumple que d ahora es la longitud del camino más corto desde u usando los vértices de $U \cup \{v\}$.

El algoritmo de Dijkstra es: tenemos una heap H sobre V, con $f = d$. Primero ponemos $d(u) = 0$, $d(v) = \infty$ para $v \neq u$, y hacemos: mientras H no es vacía, sea v el resultado de **extract-min**; por cada arista $vw \in E$ ponemos $d(w) = \text{mín}\{d(w), d(v) + w(vw)\}$ y hacemos **decrease-key** sobre w. El tiempo es el de crear la heap, hacer a lo sumo n operaciones **extract-min** y a lo sumo $2m$ operaciones **decrease-key**. Mirando directamente sobre d, **extract-min** es $O(n)$ y **decrease-key** es $O(1)$ así que en total es $O(n^2 + m) = O(n^2)$. Usando una binary heap es $O((n + m) \log n)$. Usando una Fibonacci heap es $O(m + n \log n)$.

1.1.3 Interval trees

Sea U un conjunto con un elemento $e \in U$ y $\oplus : U^2 \to U$ tal que si $a, b, c \in U$ se cumple que $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ y que $a \oplus e = e \oplus a = a$. Tenemos N elementos $a_1, \ldots, a_N$ de U y queremos mantener las operaciones $\mathbf{get}(i, j)$ (con $1 \leqq i \leqq j \leqq N$), que devuelve $a_i \oplus \cdots \oplus a_j$ y $\mathbf{set}(i, x)$, que pone $a_i = x$. Con una inicialización en tiempo $O(N)$, las dos operaciones se pueden implementar en $O(\log N)$.

Mantenemos un árbol binario que guarda intervalos y con cada intervalo $[i, j]$ guardamos $a_i \oplus \cdots \oplus a_j$. Su raíz es $[1, N]$; los hijos de $[i, j]$, con $2 \leqq j - i$, son $[i, i + \lfloor \frac{j-i}{2} \rfloor]$ y $[i + \lfloor \frac{j-i}{2} \rfloor + 1, j]$. Ahora $\mathbf{get}(i, j)$ se implementa como $\mathbf{get}(i, j, a, b)$, donde $[a, b]$ es un intervalo del árbol con $[i, j] \subset [a, b]$; si $[i, j] = [a, b]$ la respuesta está memorizada; si $[i, j] \subset [a, a + \lfloor \frac{b-a}{2} \rfloor]$ es $\mathbf{get}(i, j, a, a + \lfloor \frac{b-a}{2} \rfloor)$; si $[i, j] \subset [a + \lfloor \frac{b-a}{2} \rfloor + 1, b]$ es $\mathbf{get}(i, j, a + \lfloor \frac{b-a}{2} \rfloor + 1, b)$; en otro caso, es

$$\mathbf{get}\left(i, a + \left\lfloor \frac{b-a}{2} \right\rfloor, a, a + \left\lfloor \frac{b-a}{2} \right\rfloor\right) \oplus \mathbf{get}\left(a + \left\lfloor \frac{b-a}{2} \right\rfloor + 1, j, a + \left\lfloor \frac{b-a}{2} \right\rfloor + 1, b\right).$$

Se ve que el tiempo es $O(\log N)$. Ahora para implementar $\mathbf{set}(i, x)$ podemos hacer la misma búsqueda recursiva de manera que primero encuentra los intervalos que contienen a $[i, i]$ y luego los actualiza.

Si N es potencia de dos podemos representar el árbol como un arreglo de $2N + 1$ elementos de manera que, para $1 \leqq i \leqq N$, $A_{N+i} = a_i$ y $A_i = A_{2i} \oplus A_{2i+1}$, con $A_{2N+1} = e$. Si N no es potencia de dos extendemos $a_1, \ldots, a_N$ con $2^{\lceil \log_2 N \rceil} - N$ copias de e. Ahora $\mathbf{get}(i, j)$ es $\mathbf{get'}(i' = i + N, j' = j + N)$, que se calcula así: se pone $a = b = e$; si i' es impar se pone $a = A_{i'}$ y $i' = i + 1$; si j' es par se pone $b = A_{j'}$ y $j' = j' - 1$; ahora el resultado es

$$a \oplus \mathbf{get'}\left(\frac{i'}{2}, \frac{j'}{2}\right) \oplus b.$$

Ahora $\mathbf{set}(i, x)$ se hace así: se pone $A_{i+N} = x$ y luego $i' = \lfloor \frac{i+N}{2} \rfloor$; mientras $i' > 1$ se hace $A_{i'} = A_{2i'} \oplus A_{2i'+1}$. Es claro que las dos operaciones funcionan en tiempo $O(\log N)$. Inicializar la estructura es poner $a_1, \ldots, a_N$ en $A_{1+N}, \ldots, A_{2N}$, $A_{2N+1} = e$ y luego desde $i = N$ hasta $i = 1$ hacer $A_i = A_{2i} \oplus A_{2i+1}$. El tiempo es $O(N)$.

RMQ. Dada una lista de números $A_1, \ldots, A_n$ y $a, b \in [n]$ con $a \leqq b$, *range minimum query* es el problema: dar $\text{RMQ}(a,b) = \min\limits_{a \leqq i \leqq b} A_i$. Con un preproceso de costo $O(n)$ se puede responder una RMQ en tiempo $O(\log n)$ usando el árbol binario con la operación $\oplus = \text{mín}$ con $e = \infty$.

Con un preproceso de costo $O(n \log n)$ se puede responder una RMQ en tiempo constante. Se crean $k = \lfloor \log_2 n \rfloor$ arreglos $A^1, A^2, \ldots, A^k$, con $A^p_i = \text{RMQ}(i, i + 2^p)$ para $i \leqq n - 2^p$ gracias a la recursión $A^p_i = \text{mín}\{A^{p-1}_i, A^{p-1}_{i+2^{p-1}}\}$. Con esto, $\text{RMQ}(i,j) = \text{mín}\{A^p_i, A^p_{j-2^p}\}$, con $p = \lfloor \log_2(j - i + 1) \rfloor$.

LCA. Dado un árbol T de n vértices con raíz y vértices $u, v \in V(T)$, *lowest common ancestor* es el problema: determinar w en el camino de u a v con $d(w)$ mínimo, donde d es la distancia a la raíz. LCA se reduce a RMQ en tiempo lineal: usando BFS calculamos la función d, luego listamos los vértices en el orden en el que los recorre el DFS largado desde la raíz: tenemos $f^{-1} : [2n] \to V(T)$ que dice qué vértice visitó en cada paso y $f_-, f_+ : V(T) \to [2n]$ que dicen el primero y el último paso en el que visitó a cada vértice así que si $A : [2n] \to \mathbb{N}_0$ es $i \mapsto d(f^{-1}(i))$, $\text{LCA}(u,v) = f^{-1}(\text{RMQ}_A(f_-(u), f_+(v)))$.

RMQ se reduce a LCA. Usando el truco de la stack se pueden calcular en tiempo lineal las funciones $f_-, f_+ : [n] \to [n]$, que mapean el índice i con el índice j sujeto a $i < j$ y $\text{RMQ}(i,j) = i$ (resp. $j < i$, $\text{RMQ}(j,i) = i$, $A_j \neq A_i$) que minimiza A_j, o a -1, si no hay tales j. Creamos un árbol con vértices $[2n]$ y aristas: $i(n+i)$, con $i \in [n]$, y $(n+i)(n+j)$, con $i, j \in [n]$, $j = f_-(i)$ o $j = f_+(i)$, de manera que las hojas sean $[n]$. Ahora $\text{RMQ}(i,j) = \text{LCA}(i,j) - n$.

Bender-Farach. Con un preproceso de costo $O(n)$ se puede responder una LCA en $O(1)$. La transformamos en RMQ. Partimos el arreglo en $\frac{1}{2}\log n$ bloques de igual tamaño. Para responder RMQ en $O(1)$ basta con poder responderlo para i, j en un mismo bloque y para la unión de bloques consecutivos. Para lo primero, normalizamos los bloques restando el primer elemento a todos los números del bloque, lo cual no afecta la respuesta. Por la naturaleza del DFS, números consecutivos del bloque diferirán en 1 ó -1. Entonces hay a lo sumo $2^{\frac{1}{2}\log n - 1} = \frac{1}{2}\sqrt{n}$ posibles bloques; calculamos todas las RMQ posibles en $O(\log^2 n)$ y guardamos todo, tardando $O(\sqrt{n}\log^2 n) = O(n)$. Ahora hacemos una pasada de $O(n)$ e identificamos los bloques, así que ya sabemos responder RMQ con i, j en el mismo bloque en $O(1)$. Ahora tomamos el mínimo elemento de cada bloque y formamos un nuevo arreglo con estos $2n/\log n$ elementos. Con un precómputo de $O(2n/\log n \log(2n/\log n)) = O(n)$ vimos que podemos responder RMQ en $O(1)$, que es lo que faltaba.

Irónicamente, para conseguir la misma complejidad sobre RMQ va a haber que transformarlo en LCA y transformar éste de vuelta en RMQ.

Interval trees en 2D. Aprendimos a responder queries sobre los puntos en un intervalo $[x_0, x_1]$ en tiempo $O(\log n)$. Podemos extender esto a queries sobre rectángulos $[x_0, x_1] \times [y_0, y_1]$ en tiempo $O(\log^2 n)$: un interval tree sobre x parte el rectángulo en $O(\log n)$ rectángulos $[x_i, x_{i+1}] \times [y_0, y_1]$. Partimos cada uno de estos rectángulos en $O(\log n)$ rectángulos $[x_i, x_{i+1}] \times [y_j, y_{j+1}]$, sobre los cuales la query está precomputada, y son $O(n \log n)$ en total. Además cada punto está en $O(\log^2 n)$ rectángulos así que en ese tiempo se puede modificar el dato asociado a un punto.

Supongamos que sólo queremos responder la query cuántos puntos hay en un rectángulo $[x_0, x_1] \times [y_0, y_1]$. Con lo anterior se responde fácilmente en $O(\log^2 n)$ dado un precómputo de tiempo $O(n \log n)$. Se pueden bajar las queries a $O(\log n)$. La idea es hacer una sola búsqueda binaria para los y. En cada intervalo $[i, j]$, que representa puntos con x consecutivos, guardamos los valores de y de los puntos comprendidos ordenados y, para cada uno, su posición (para el primero, 1, para el segundo, 2, y así), un puntero que apunta al primer y mayor o igual a él en el intervalo $[i, \lfloor \frac{i+j}{2} \rfloor]$ y otro que apunta al primero mayor o igual a él en $[\lfloor \frac{i+j}{2} \rfloor + 1, j]$.

Entonces cuando vayamos recorriendo el árbol de intervalos podemos saber dónde empieza y dónde termina nuestra ventana en y; cuando llegamos a un intervalo que no vamos a partir, sabemos las posiciones de nuestros y, así que también cuántos hay entre ellos. Eso es todo.

1.1.4 2,4-trees y splay trees

Un *2,4-tree* es un árbol $T = (U, E)$ con raíz tal que sus hojas u guardan un número $f(u)$, sus nodos internos (los que no son una hoja) tienen 2, 3 ó 4 hijos, las distancias de la raíz a las hojas es siempre la misma y tal que si u es un nodo interno sus hijos v_1, v_2, v_3 (si existe) y v_4 (si existe) cumplen que los números de las hojas de v_i son menores que los de v_{i+1}. Veamos que se pueden implementar sobre esta estructura las operaciones **access**(x, T), que, dado T y $x \in \mathbb{R}$ determina si hay una hoja $u \in T$ con $f(u) = x$ y si la encuentra la devuelve, **insert**(x, T), que inserta una nueva hoja u con $f(u) = x$, **delete**(x, T), que determina si hay una hoja u con $f(u)$ y si la encuentra la borra, **join**(T_1, T_2), que, asumiendo que los reales asignados a las hojas de T_1 son todos menores a los asignados a las hojas de T_2, destruye T_1 y T_2 y devuelve un nuevo 2,4-tree que contiene las hojas de T_1 y las de T_2, y **split**(T, x), que destruye T devolviendo dos árboles, T_1 y T_2, tales que si u es hoja de T_1 y v es hoja de T_2 se cumple $f(u) \leqq x < f(v)$, todas en tiempo $O(\log n)$, donde n es la cantidad de hojas que contiene el árbol.

Cada nodo interno u guarda enlaces a sus hijos y, para cada uno, el número más grande de sus hojas. Notemos que la altura es $O(\log n)$, así que la operación **access**(x, T) se puede implementar fácil en $O(\log n)$. Para hacer **insert**(x, T), buscamos el nodo u (único) tal que x tiene que ser hijo de u; si tiene dos o tres hijos, agregamos uno con valor x; si no, agregamos un nuevo nodo u', repartimos los cinco nodos (los cuatro hijos de u y el nuevo): los primeros tres en u y los segundos dos en u'; hacemos esto recursivamente: ahora el problema es insertar u' en el padre de u; cuando llegamos a la raíz, si tiene grado cuatro agregamos una nueva raíz, con lo que aumenta la altura del árbol en uno. De nuevo el tiempo es $O(\log n)$. Para hacer **delete**(x, T) buscamos el nodo y lo borramos; esto puede causar que su padre u ahora tenga un solo hijo; si un hermano v del padre tiene tres o cuatro hijos, le pasamos un hijo de v a u; si un hermano tiene dos o tres hijos, eliminamos u y le pasamos su único hijo; esto puede causar que el padre de u ahora tenga un solo hijo, pero podemos proceder recursivamente; si al final llegamos a que la raíz tiene un solo hijo, la eliminamos y hacemos que su hijo sea la nueva raíz. De nuevo, el tiempo es $O(1)$ por nivel, así que es $O(\log n)$.

La operación **join**(T_1, T_2) se hace así: sean h_1 y h_2 las alturas; si $h_1 = h_2$, ponemos un nodo raíz, lo conectamos a las raíces de T_1 y T_2 y estamos; si $h_1 > h_2$, vamos recorriendo T_1 desde la raíz hasta que encontramos un nodo u cuya distancia a las hojas es $h_2 + 1$, siempre bajando por el último hijo (el tercero o el segundo); la tarea se reduce a insertar en u la raíz de T_2, que vimos cómo hacerlo con **insert**. El caso $h_1 < h_2$ es similar. El tiempo es $O(|h_1 - h_2| + 1)$, que por supuesto es $O(\log n)$, donde n es la cantidad total de hojas. Veamos la operación **split**(T, x), la más difícil. Buscamos x guardando en dos stacks los nodos cuyos hijos son menores o iguales que x y, en otro, los nodos cuyos hijos son mayores que x que nos vamos encontrando al buscar x. Al final tomamos cada stack, hacemos join con sus elementos y devolvemos los dos árboles resultantes. Es claro que es correcto. Veamos que el tiempo es $O(\log n)$. Los subárboles que vamos a unir tienen alturas no decrecientes $h_1 \leqq h_2 \leqq \cdots \leqq h_k$; las alturas de los árboles intermedios que genera **join** son $h'_2 \leqq \cdots \leqq h'_k$. Se ve que $h'_i \leqq h_i + 1$ para $i = 2, \ldots, k$: Para cada altura dada hay a lo sumo tres h_i. Si tomamos el primero, cuando le hacemos join es con un árbol de como mucho su altura por hipótesis inductiva. Si su altura crece entonces deja lugar para los otros dos de su altura (por cómo es **insert**), y vale $h'_i \leqq h_i + 1$ también para ellos. Ahora la cota $O(\log n)$ es clara.

Teorema (Hopcroft). *Usando 2,4-trees podemos mantener un conjunto de elementos ordenados con operaciones* ***access****,* ***insert****,* ***delete****,* ***join*** *y* ***split*** *en tiempo logarítmico.*

Podemos decir más. El costo de hacer m operaciones **insert** o **delete** sobre un 2,4-tree vacío cuesta lo que toma ubicar los nodos a ser insertados o borrados más lo que toma modificar la estructura, y esto último cuesta $O(1)$ amortizado. Si d_2, d_3, d_4 son las cantidades de nodos con 2, 3 y 4 hijos, respectivamente, definimos el potencial Φ como $d_2 + d_3 + 3d_4$. Se ve que si c es la cantidad de modificaciones de un **insert** o un **delete** entonces $c + \Phi' - \Phi \leqq 1$, donde Φ' es el nuevo potencial. En efecto, lo que cuesta en el **insert** es partir un nodo de 4 hijos en dos, uno de 2 y uno de 3, lo cual hace $\Phi' - \Phi = -1$, y lo que cuesta en el **delete** es unir dos nodos con 2 hijos en uno de 3, lo cual también hace $\Phi' - \Phi = -1$. Entonces el costo total es $c_1 + \cdots + c_m \leqq c_1 + \cdots + c_m + \Phi_m - \Phi_0 = \sum_{i=1}^{m}(c_i + \Phi_i - \Phi_{i-1}) \leqq m$, como queríamos. Ahora en cada nodo pongamos dos punteros, uno al nodo de su misma altura que está a su izquierda y otro al de su derecha (pueden no ser sus hermanos). Se ve que se puede incorporar a las operaciones sin problemas. Ahora si tomamos una hoja, la operación **access** se puede hacer en tiempo $O(\log d)$, si la hoja buscada dista a lo sumo d de la que partimos en la lista lineal de las hojas: en efecto, subimos hacia la raíz mientras no sea descendiente del nodo que estamos mirando ni del de su derecha ni su izquierda; cuando sí es descendiente, bajamos hacia la hoja; en cada paso avanzamos al menos 2^h hojas hacia el objetivo (h altura), así que en total hicimos $O(\log d)$ pasos. Cambiando la parte de búsqueda de **insert** y **delete** por este método cuestan $O(\log d)$ más $O(1)$ amortizado. Entonces tenemos:

Teorema (Huddleston-Mehlhorn). *Dado un 2,4-tree y un puntero a una hoja, el finger, podemos hacer* ***access****,* ***insert*** *y* ***delete*** *de números a distancia d del finger en tiempo $O(\log d)$ amortizado.*

Un *splay tree* es una estructura que soporta las mismas operaciones que la anterior, con los mismos tiempos, pero amortizados. Es un árbol con raíz, binario, con números en todos los nodos, de manera que el hijo izquierdo de un nodo tiene un número menor que él y el derecho tiene uno mayor. Se basa sobre una operación, **splay**(u), que logra estructurarlo. Lo que hace es tomar el vértice e ir haciendo *rotaciones* hasta llevarlo a la raíz. Una rotación consiste en tomar un vértice v con hijos A y B y su padre w con hijos v y C y poner v en lugar de w, poner A como su hijo izquierdo, w como su derecho, B como el hijo izquierdo de w y C como el hijo derecho; o lo simétrico, si v es el hijo derecho de w. Lo que hace **splay**(u) es: mientras u tiene padre, si u y $p(u)$ son los dos hijos izquierdos o derechos, rotamos en $p^2(u)$ y luego en $p(u)$; si uno es izquierdo y el otro derecho, rotamos en $p(u)$ y luego en el nuevo padre de u; si $p(u)$ no tiene padre simplemente rotamos en $p(u)$.

La operación **access** ubica el nodo u con $f(u) = x$ y hace **splay**(u); **insert** busca el lugar donde tendría que ir x, lo agrega y hace **splay**; **delete** busca u; si tiene los dos hijos, va a su hijo izquierdo y luego baja por hijos derechos hasta que no puede más; este nodo, v, que es el que tiene $f(v) < f(u)$ máximo, se intercambia por u; ahora si u tiene un hijo, se lo cambia por el hijo, se lo elimina y se hace **splay** con su padre (que es el padre del antecesor v, si se intercambiaron, o del u original). Hacer **join** es buscar el máximo del primer árbol, quitarlo, ponerlo de raíz, y colgarle los árboles como hijos. Para hacer **split** buscamos u, hacemos **splay**, con lo que pasa a ser la raíz, y devolvemos su hijo izquierdo y su hijo derecho.

Veamos ahora la cota $O(m \log n)$ del tiempo total, donde m es la cantidad de operaciones y n es la cantidad total de elementos insertados. Ponemos pesos no negativos w a los vértices, definimos $s(v)$ como la suma de los pesos del vértice v y sus descendientes, $r(v) = \log s(v)$ y hacemos que Φ, el potencial, sea $\sum_{v \in V} r(v)$. Veamos que si se hace una rotación en un nodo x con padre y entonces el cambio de potencial $\Phi' - \Phi$ es a lo sumo $r'(x) + r'(y) - r(x) - r(y) \leqq 3(r'(x) - r(x))$ (porque $r'(x) \geqq r(x)$ y $r'(y) \leqq r(y)$). Si x es hijo izquierdo de y, y es hijo izquierdo de z, y primero rotamos en z y luego en y, el cambio de potencial es

$$r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) = r'(y) + r'(z) - r(x) - r(y) \leqq$$

$$\leqq r'(x) + r'(z) - 2r(x) = (r(x) - r'(x)) + (r'(z) - r'(x)) + 3(r'(x) - r(x)) =$$
$$= \log \frac{s(x)}{s'(x)} + \log \frac{s'(z)}{s'(x)} + 3(r'(x) - r(x)) \leqq 2 \log \frac{s(x) + s'(z)}{2s'(x)} + 3(r'(x) - r(x)) \leqq$$
$$\leqq 2 \log \frac{s'(x)}{2s'(x)} + 3(r'(x) - r(x)) = -2 + 3(r'(x) - r(x)).$$

Similarmente se ve que el cambio de potencial al hacer la otra doble rotación también es a lo sumo $-2 + 3(r'(x) - r(x))$. Entonces al hacer una operación **splay** sobre v el cambio de potencial es a lo sumo $1 - c + 3(r'(v) - r(v))$, donde c es la cantidad de rotaciones. Si ponemos $w = 1$, como v al final es la raíz, $r'(v) = \log n$, así que el cambio de potencial es a lo sumo $1 - c + 3 \log n$. El costo de las operaciones **splay** domina la complejidad, y es (uno por cada operación) $c_1 + \cdots + c_m \leqq c_1 + \cdots + c_m + \Phi_m - \Phi_0 = \sum_{i=1}^{m} (c_i + \Phi_i - \Phi_{i-1}) \leqq \sum_{i=1}^{m} (c_i + 1 - c_i + 3 \log n) = m + 3m \log n = O(m \log n)$, como queríamos.

Ahora supongamos que cada nodo x es accedido $t(x)$ veces. En las cuentas del párrafo anterior pongamos $w(x) = t(x)$. El cambio de potencial por splay sería $1 - c + 3(r'(x) - r(x)) \leqq 1 - c + 3(\log(\sum_{i=1}^{n} t(x_i)) - \log t(x)) = 1 - c + 3(\log m/t(x))$. El tiempo total sería, pues, $O(\sum_{i=1}^{n} t(x_i) \log(m/t(x_i)))$. Si el árbol fuera óptimo, pero estático, el tiempo total sería el mismo salvo una constante multiplicativa.

Teorema (Sleator-Tarjan). *Usando un splay tree podemos mantener un conjunto de elementos ordenados con operaciones* ***access***, ***insert***, ***delete***, ***join*** *y* ***split*** *en tiempo logarítmico amortizado, que además toma el mismo tiempo que un árbol binario estático óptimo construído para la distribución de los accesos, salvo una constante multiplicativa*[1].

1.1.5 Border y suffix array

Si s es una string con $|s| = n$ queremos encontrar su *border array*: números $b_1, \ldots, b_n$ donde b_i cumple que $s_1 \ldots s_{b_i} = s_{i-b_i+1} \ldots s_i$ y b_i es máximo, o sea b_i es la medida del máximo prefijo propio igual a un sufijo en la substring $s_1 \ldots s_i$. Los números b_i se calculan en tiempo $O(n)$: vamos desde $i = 1$ hasta $i = n$, con $j = b_i$; si $s_j = s_{i+1}$ entonces $b_{i+1} = j + 1$ y avanzamos j; la siguiente chance de prefijo igual a sufijo va a ser $b_{b_i} + 1$, y así retrocediendo; en cada iteración j avanza como mucho uno y en total no decrece más de lo que avanzó, que es a lo sumo n en total, así que el tiempo es $O(n)$.

Si s es una string con $|s| = n$, sobre el alfabeto $[n]$, se puede encontrar el suffix array de s en tiempo $O(n)$ (Kärkkäinen-Sanders). Sea s_i el i-ésimo caracter de s y s^i el sufijo de s empezando en el caracter i. El suffix array de s es un map $SA : [n] \to [n]$ tal que $(s^{SA_i})_{i \in [n]}$ está ordenado.

Suffix-Array(s)

1. Hago que n sea múltiplo de tres, agrego dos ceros al final.
2. Ordeno $(s_i s_{i+1} s_{i+2})_{3 \nmid i}$ en tiempo lineal con radix sort. Esto da $f : [\frac{2n}{3}] \to [n]$ que dice el número de orden de $s_i s_{i+1} s_{i+2}$. Ternas iguales tienen que dar igual.
3. Pongo $s' = f_1 f_4 \ldots f_{n-2} f_2 f_5 \ldots f_{n-1}$. Hago que a sea Suffix-Array(s').
4. Recupero de a el orden de $(s^i)_{3 \nmid i}$. Hago que f_i sea $3a_i + 1$ si $a_i \leqq \frac{n}{3}$ y $3(a_i - \frac{n}{3}) + 2$ si no.
5. Ahora el orden de $(s^i)_{3|i}$ es el de $(s_i f_{i+1}^{-1})_{3|i}$, que sale haciendo radix sort. Esto da $g : [\frac{n}{3}] \to [n]$ con $(s^{g_i})_{i \in [\frac{n}{3}]}$ es $(s^i)_{3|i}$ ordenado.
6. Ahora hay que mergear f y g para obtener SA. Esto se reduce a comparar s^i, $3 \mid i$, con s^j, $3 \nmid i$. Si $s_i \neq s_j$ es claro. Si no, si $3 \mid j - 1$ comparamos s^{i+1} y s^{j+1} usando f^{-1}; si $3 \mid j - 2$, comparamos s_{i+1} con s_{j+1} y, si no basta, s^{i+2} con s^{j+2}, de nuevo con f^{-1}.

[1] Hay otros resultados llamativos sobre el desempeño de la operación splay. Cole demostró que el costo de splay sobre un nodo que está a distancia d de la raíz (en el orden lineal) es $O(\log d)$ amortizado, lo cual vuelve al splay tree tan bueno como un 2,4-tree con un finger.

La complejidad satisface $T(n) \leqq T(\frac{2}{3}n) + O(n)$, por lo que es $O(n)$. Veamos que con el suffix array se puede obtener el longest common prefix array $Lcp : [n-1] \to [n]$, que cumple que $Lcp[i]$ es la longitud del prefijo común más largo entre s^{SA_i} y $s^{SA_{i+1}}$, en tiempo $O(n)$. Los calculamos en orden $SA_1^{-1}, \ldots, SA_n^{-1}$. El primero lo hacemos a mano. Si tenemos el i, sea $j = SA_i^{-1} + 1$; si $Lcp[SA_i^{-1}] = l$ sabemos que $s_i^{i+l} = s_j^{j+l}$ y que $s_{i+l+1} \neq s_{j+l+1}$, así que $Lcp[SA_{i+1}^{-1}] \geqq l - 1$. Entonces para sacar este nuevo valor recorremos s desde s_{i+1+l} y $s_{SA_{i+1}^{-1}+1+l}$. Se ve claro que el tiempo es $O(n)$.

El prefijo más largo entre dos sufijos s^i, s^j es el mínimo de $Lcp[k]$, con k entre SA_i^{-1} y SA_j^{-1}. Se puede determinar el substring común de dos strings s_1, s_2 de longitud máxima en tiempo lineal. También se pueden encontrar todos los substring palíndromos maximales en tiempo $O(n)$[2].

1.1.6 Ejercicios

1. Dada una secuencia $a_1, \ldots, a_n \in [n]$ encontrar todas las subsecuencias $a_i, a_{i+1}, \ldots, a_j$ que son una permutación de $\{1, \ldots, j - i + 1\}$ en tiempo $O(n)$.[3]

2. (Sliding window) Dados números $a_1, \ldots, a_N$ y un número $k \leqq N$, devolver todos los números $\min\limits_{i \in [j, j+k)} a_i$ para $j = 1, \ldots, N + 1 - k$ en tiempo $O(N)$.

3. (Javier Corti) Dados números $a_1, \ldots, a_n$ contar la cantidad de pares $i < j$ con $a_i, a_{i+1}, \ldots, a_j \leqq \min\{a_i, a_j\}$ en tiempo $O(n)$.

4. Dados números $a_1, \ldots, a_n$ determinar $\max\limits_{1 \leqq i < j \leqq n} \left\{ (j - i + 1) \min\limits_{i \leqq k \leqq j} a_k \right\}$ en tiempo $O(n)$.[4]

5. Dada una matriz $A \in \{0, 1\}^{m \times n}$ encontrar $1 \leqq a < b \leqq m$ y $1 \leqq c < d \leqq n$ tales que $A_{ij} = 0$ para $(i, j) \in [a, b] \times [c, d]$ y $(b - a + 1)(c - d + 1)$ es máximo.

6. (Grafo Euleriano) Un ciclo euleriano (resp. camino) es un ciclo (resp. camino) no necesariamente simple pero que pasa por cada arista exactamente una vez. Demostrar que un grafo G tiene un ciclo euleriano (resp. camino) si y sólo si el grado de todo vértice es par (resp. todos menos dos). Dar un análogo para grafos dirigidos. Dar un algoritmo para encontrar un ciclo o camino euleriano si hay en tiempo lineal.

7. Dado un grafo G determinar si es bipartito en tiempo lineal.

8. Dado un grafo dirigido G buscar un ciclo de longitud impar en tiempo lineal.[5]

9. (Articulation points) Si G es no dirigido, $u \in V$ (resp. $uv \in E$) se dice *punto de articulación* (resp. *puente*) si al eliminarlo aumenta la cantidad de componentes conexas del grafo. Dado G encontrar todos sus puntos de articulación y todos sus puentes en tiempo lineal.[6]

---

[2] Calculamos el suffix array de la string que viene de pegar s, un caracter no usado, y s invertida. Probamos todos los centros posibles para los palíndromos: a ambos lados se van a extender tanto como el longest commmon prefix entre el sufijo y el prefijo invertido.

[3] Sugerencia. Notar que $a_i, a_{i+1}, \ldots, a_j$ es permutación de $\{1, \ldots, j - i + 1\}$ sii son todos diferentes, tiene un uno, que sea a_k, y el máximo es $j - i + 1$; dos casos: el máximo está en $a_1, \ldots, a_{k-1}$ y es M, en cuyo caso $j = i + M - 1$, o el máximo está en $a_{k+1}, \ldots, a_j$, que es simétrico.

[4] Sugerencia. Para cada $k \in [n]$ buscamos i_k mínimo y j_k máximo tales que $i_k \leqq k \leqq j_k$ y $a_k = \min\limits_{i_k \leqq l \leqq j_k} a_l$; la respuesta es $\max\limits_{k \in [n]} (j_k - i_k + 1)a_k$. Buscar i_k y j_k por separado.

[5] Sugerencia. Primero asumir que es fuertemente conexo.

[6] Sugerencia. Notar que la raíz del DFS es articulation sii tiene más de un hijo y un vértice no-raíz v es articulation sii tiene un hijo cuyos descendientes (él incluído) no tienen backedges a un ancestro de v. Calcular $low(u)$: el mínimo entre $pre(u)$ y $pre(w)$, donde vw es un backedge de un descendiente de u.

10. (Biconnected components) Si G es no dirigido definimos la relación de equivalencia $\sim \subset E^2$ dada por $e \sim e'$ si y sólo si hay un ciclo simple que contiene tanto a e como a e'. Determinar las clases de equivalencia (las componentes biconexas) en tiempo lineal.

11. (2SAT) Dadas variables $x_1, \ldots, x_n$ y cláusulas $c_1, \ldots, c_m$, donde una cláusula tiene la forma $x_i \vee x_j$, $x_i \vee \neg x_j$ o $\neg x_i \vee \neg x_j$, determinar si se le puede asignar valores de verdad a las variables para que las cláusulas resulten todas verdaderas.[7]

12. (BFS con varias queues) Supongamos que tenemos un grafo y k tipos de aristas. Encontrar la longitud del menor camino de un vértice dado a todos los demás, donde en los caminos no se permite que dos aristas consecutivas sean del mismo tipo, en tiempo $O(nk+m)$.

13. A la binary heap agregarle las operaciones **minimum** en $O(1)$ e **insert**, **increase-key** y **delete** en $O(\log n)$.

14. (d-ary heap) Usando un arreglo como con la binary heap se puede implementar una d-ary heap, que es igual salvo que todo nodo tiene d hijos (salvo los hijos y a lo sumo uno). Mostrar cómo implementar las operaciones **minimum** en $O(1)$, **extract-min**, **increase-key** y **delete** en $O(d \log_d n)$ y **insert** y **decrease-key** en $O(\log_d n)$.

15. A la Fibonacci heap agregarle las operaciones **minimum** en $O(1)$, **insert** en $O(1)$ amortizado e **increase-key**, **delete** y **merge** (dadas otra Fibonacci heap agregar todos sus nodos) en $O(\log n)$ amortizado.

16. Modificar la Fibonacci heap para que soporte operaciones **merge** en tiempo $O(1)$ amortizado.[8]

17. (Prim) Dado un grafo no dirigido G conexo con pesos en las aristas, encontrar un árbol (V, E') con $E' \subset E$ y suma de pesos en las aristas mínimo, en tiempo $O(m + n \log n)$.

18. Dado un grafo no dirigido G conexo con pesos en las aristas computar la tabla $A \in \mathbb{R}^{V \times V}$ que en A_{uv} tiene el mínimo, entre todos los caminos de u a v, de la arista más pesada del camino, en tiempo $O(n^2 + m \log m)$.

19. (Agustín) Describir una estructura de datos que mantenga n números reales $a_1, \ldots, a_n$ y que soporte las operaciones: **get**(i, j), que devuelve el mínimo de $\{a_i, \ldots, a_{j-1}\}$, **set**(i, x), que hace $a_i = x$ y **sum**(i, j, x), que suma x a todos los números $a_i, \ldots, a_{j-1}$, todas en tiempo logarítmico, con inicialización dados $a_1, \ldots, a_n$ en tiempo lineal.

20. Describir una estructura de datos que, dada $\oplus : U^2 \to U$ asociativa con elemento neutro e y un natural N, mantenga N elementos de U, algunos de los cuales están activos y otros no, con las operaciones **get**(i, j), que devuelve $a_i \oplus \cdots \oplus a_j$, donde a_k es el k-ésimo elemento de la secuencia si está activo o e si no, **set**(i, x), que hace que el i-ésimo elemento sea x, y **activate**(i, j) y **deactivate**(i, j), que activa (resp. desactiva) los elementos en el intervalo $[i, j]$, todas en tiempo $O(\log N)$.

21. (Rectangle union) Dados n rectángulos en el plano con lados paralelos a los ejes determinar el área de su unión en tiempo $O(n \log n)$.

---

[7] Sugerencia. Reducir el problema a encontrar las componentes fuertemente conexas del grafo que tiene por vértices a las variables y sus negadas y por aristas a las dos implicaciones $\neg x \Rightarrow y$ y $\neg y \Rightarrow x$ equivalentes a $x \vee y$.

[8] Sugerencia. En lugar del arreglo b guardamos una lista con las raíces. Para amortizar hacemos que los **repair** se hagan todos juntos cuando se haga un **extract-min**, de manera que **merge** sea simplemente hacer **merge** de listas. Notar que la cantidad de **repair** totales se mantiene; hay que agregar una operación interna que hace que las raíces pasen a cumplir **F3** en tiempo $O(\log n)$ (aparte de los **repair**): armamos un arreglo $b : \{0, \ldots, \lfloor 2 \log_2(|U| + 2) \rfloor\} \to U$ (ya que $F_{\deg^+(u)+1} \leqq n$) y vamos metiendo las raíces de la lista; cuando dos colisionan, hacemos **repair** y seguimos.

22. See $A \subset \mathbb{R}^2$ un conjunto de n puntos en el plano y $f : A \to \mathbb{R}$ una función. Dados q puntos $(x_1, y_1), \ldots, (x_q, y_q)$ calcular los números

$$\sum_{\substack{u \in A \\ u_x \geqq x_i, u_y \geqq y_i}} f(u)$$

para $i = 1, \ldots, q$ en tiempo $O((n+q)\log(n+q))$[9].

23. Dado un conjunto A de n puntos en $\mathbb{R}^2$ y otro, Q, de q puntos, calcular, para cada punto de Q, la distancia al punto de A más cercano, donde la distancia entre (x_1, y_1) y (x_2, y_2) es $|x_1 - x_2| + |y_1 - y_2|$, en tiempo $O((n+q)\log(n+q))$.

24. Modificar el 2,4-tree y el splay tree para que guarde en cada nodo la cantidad de (hojas) descendientes, de manera que los tiempos de las operaciones sigan siendo logarítmicos pero la estructura pueda devolver el k-ésimo elemento (en orden) en tiempo logarítmico (amortizado).

25. Modificar el 2,4-tree y el splay tree para que soporte la operación **reverse**(u, v), que invierte el orden de todos los elementos contenidos entre u y v en tiempo logarítimico.

26. Tanto el 2,4-tree como el splay tree permiten hacer **merge** y **difference** de listas ordenadas, que devuelven su unión y diferencia, respectivamente, en tiempo $O(m\log(n/m))$, donde m, n $(m \leqq n)$ son sus longitudes[10].

27. (Knuth-Morris-Pratt) Dada una string s y una t encontrar todos índices i tales que $t_i \ldots t_{i+|s|-1} = s$ en tiempo $O(|s| + |t|)$.

28. Dada una string s con $n = |s|$ determinar en tiempo $O(n)$ los números $p_1, \ldots, p_n$ donde $p_i \mid i$ cumple que $s_1 \ldots s_i = (s_1 \ldots s_{p_i})^{i/p_i}$ y es mínimo.

29. Dadas una string s de tamaño n y strings $s_1, \ldots, s_k$ cuya longitud total es m, determinar para cada una si es substring o no de s en tiempo total $O(n+m)$. Con un preproceso de tiempo $O(n)$ determinar si una string de longitud m es substring de s en tiempo $O(m + \log n)$.

30. Dada una string s contar la cantidad de substrings distintas en tiempo lineal.

31. (Agustín Gutiérrez) Dado un conjunto de n puntos dar todas las rectas tales que, para todo punto del conjunto, su simétrico con respecto a la recta también está en el conjunto, en tiempo $O(n\log n)$.[11]

32. (Pablo Heiber) Dadas strings $s_1, \ldots, s_n$ devolver todos los pares de strings s_i, s_j tales que s_i es substring de s_j en tiempo $O(\sum_{i=1}^{n} |s_i| + n\log n + t)$, donde t es la cantidad de esos pares.

33. Dada una string s y un número m calcular la substring más larga de s que aparece repetida al menos m veces en s en tiempo $O(|s|)$.

---

[9] Sugerencia. Mantener un árbol de intervalos para las sumas de $f(u)$ de los puntos en orden de x. Luego tomar los puntos iniciales y los puntos de query, ordenarlos por y en orden decreciente, e ir agregando los puntos al árbol y procesando las queries.

[10] Sugerencia. Se inserta la lista chica en la grande. En el 2-4-tree usar el último elemento insertado como finger.

[11] Sugerencia. Separar los puntos según su distancia al baricentro; encontrar los ejes de simetría en las circunferencias es buscar palíndromos en el array de las distancias; el resultado es la intersección de todas estos conjuntos de ejes de simetría.

1.2 Técnicas algorítmicas

1.2.1 Divide-and-conquer

La idea es: dado un problema, partirlo en subproblemas, resolverlos recursivamente y combinar las soluciones de los subproblemas para obtener una solución al problema original. La complejidad se calcula resolviendo una recursión de la forma $T(O(1)) = O(1)$, $T(n) = a_1T(n_1) + \cdots + a_kT(n_k) + f(n)$, donde $n_1 + \cdots + n_k = n$ y $f(n)$ es el tiempo que toma combinar los subproblemas.

Mergesort. Si queremos ordenar un arreglo de n elementos $a_1, \ldots, a_n$, ordenamos $a_1, \ldots, a_{n/2}$, luego $a_{n/2+1}, \ldots, a_n$ y al final hacemos *merge* con las dos listas ordenadas para obtener la concatenación ordenada. Esto se puede hacer en $O(n)$: miramos el primer elemento de la primera lista, si es menor al primero de la segunda, va a ser el primero de la lista total; si no, es el primero de la segunda; seguimos así sacando elementos de las dos listas hasta que no quedan más elementos por sacar. El tiempo total del algoritmo cumple $T(1) = O(1)$ y $T(n) = 2T(n/2) + O(n)$, así que es $O(n \log n)$.

Strassen. Queremos multiplicar dos matrices $A, B \in \mathbb{R}^{n\times n}$. Descomponemos A y B en submatrices cuadradas de igual tamaño así:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}.$$

Tenemos claramente

$$AB = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}.$$

Y se comprueba trivialmente que poniendo

$$\begin{aligned} P_1 &= a(f - h) \\ P_2 &= (a + b)h \\ P_3 &= (c + d)e \\ P_4 &= d(g - e) \\ P_5 &= (a + d)(e + h) \\ P_6 &= (b - d)(g + h) \\ P_7 &= (a - c)(e + f) \end{aligned}$$

tenemos

$$AB = \begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_5 + P_1 - P_3 - P_7 \end{pmatrix}.$$

Así que con siete multiplicaciones de matrices de la mitad del tamaño logramos multiplicar las dos matrices originales. Esto da un algoritmo de tiempo $T(1) = O(1)$, $T(n) = 7T(n/2)+O(n^2)$, que es $T(n) = O(n^{\log_2 7})$.

Fast Fourier Transform. Queremos multiplicar dos polinomios $p, q \in \mathbb{C}[x]$ de grado n. El grado de pq es a lo sumo $2n$ así que si los evaluamos en las $m = 2n + 1$ raíces de la unidad $1, \omega, \omega^2, \ldots, \omega^{m-1}$, donde $\omega = \cos(\frac{2\pi}{m}) + i\sin(\frac{2\pi}{m})$, $(pq)(\omega^i) = p(\omega^i)q(\omega^i)$, así que podemos obtener $pq = a_{m-1}x^{m-1} + \cdots + a_1x + a_0$ usando

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{m-1} \end{pmatrix} = \frac{1}{m} \begin{pmatrix} 1 & 1 & \ldots & 1 \\ 1 & \bar{\omega} & \ldots & \bar{\omega}^{m-1} \\ \vdots & \vdots & & \vdots \\ 1 & \bar{\omega}^{m-1} & \ldots & \bar{\omega}^{(m-1)^2} \end{pmatrix} \begin{pmatrix} (pq)(1) \\ (pq)(\omega) \\ \vdots \\ (pq)(\omega^{m-1}) \end{pmatrix} = \begin{pmatrix} r(1) \\ r(\omega^{m-1}) \\ \vdots \\ r(\omega^{m-(m-1)}) \end{pmatrix},$$

donde $r = \frac{1}{m}(pq)(\omega^{m-1})x^{m-1} + \cdots + \frac{1}{m}(pq)(\omega)x + \frac{1}{m}(pq)(1)$. Así que el problema se reduce a evaluar polinomios en raíces de la unidad.

Supongamos que p es un polinomio de grado $n = 2^m - 1$ y queremos evaluarlo en las 2^m-ésimas raíces de la unidad $1, \omega, \ldots, \omega^n$. Notemos que $p(x) = p_1(x^2)x + p_2(x^2)$, donde p_1 es $\sum_{i=0}^{(n+1)/2} a_{2i+1}x^i$ y p_2 es $\sum_{i=0}^{(n+1)/2} a_{2i}x^i$, así que el problema se reduce a calcular p_1 y p_2 sobre las 2^{m-1}-ésimas raíces de la unidad $1, \omega^2, \ldots, \omega^{2(2^{m-1}-1)}$, que se puede hacer recusivamente. La complejidad es $T(1) = 1$, $T(n) = 2T(n/2) + O(n)$, que da $O(n \log n)$.

Con mucho cuidado se puede eliminar la recursión y reducir la memoria de $\Theta(n \log n)$ a $\Theta(n)$. Si a es el vector con los coeficientes $a[0], \ldots, a[n-1]$, ponemos la permutación A dada por $A[\mathrm{rev}(k)] = a_k$, donde $\mathrm{rev}(k)$ es el número que tiene los dígitos binarios de k pero en el orden inverso. Ahora vamos desde $m = 2$ hasta $m = n$, con $\omega_m = \cos(\frac{2\pi}{m}) + i\sin(\frac{2\pi}{m})$; en cada paso vamos desde $k = 0$ hasta $k = n - 1$ sumando m; en cada paso ponemos $\omega = 1$ y vamos desde $j = 0$ hasta $m/2 - 1$ y ponemos $t = \omega A[k + j + m/2]$, $u = A[k + j]$, $A[k + j] = u + t$, $A[k + j + m/2] = u - t$ y $\omega = \omega\omega_m$. Se comprueba que al final A es p evaluado en las raíces de la unidad.

1.2.2 Programación dinámica

Supongamos que un problema se puede resolver recursivamente. Una posibilidad es tomar el problema, partirlo en subproblemas y resolver los subproblemas independientemente; recorremos el árbol de la recursión desde la raíz hasta las hojas. Otra posibilidad es resolver los subproblemas en el orden inverso: empezamos por las hojas y vamos subiendo. Esta es la idea de la programación dinámica como la vamos a ver acá. Quizás el ejemplo más claro es el de calcular los números de Fibonacci: si usamos la recursión $f_{n+2} = f_{n+1} + f_n$ tardamos f_n operaciones, que es exponencial en n; si vamos calculando $f_3, f_4, \ldots, f_n$ en ese orden el tiempo se reduce a lineal en n.

La programación dinámica se aplica en problemas de optimización a través de lo que se llama principio de optimalidad. Un problema es de subestructura óptima si una solución óptima contiene soluciones óptimas a subproblemas. Si encontramos que un problema cumple el principio de optimalidad ya sabemos que podemos dar una solución con programación dinámica. La idea es: dado un problema, encontrar uno más general tal que la solución al problema inicial contenga una subestructura óptima en su problema particular y podamos reducir la tarea de encontrar una solución del problema inicial a optimizar la subestructura. Un ejemplo muy claro es el del camino mínimo: todo subcamino de un camino mínimo debe ser mínimo, porque si hubiera un subcamino más corto podríamos reemplazarlo.

En la práctica estas ideas cristalizan en el siguiente método: dado un problema lo generalizamos a otro, buscamos una recursión para resolver el problema más general, definimos los casos base, buscamos una manera de recorrer todos los subproblemas en orden y, si es requerido, buscamos la manera de recuperar la estructura de la solución que encontramos explícitamente.

Subsumas. Dada $A \in \mathbb{R}^{m \times n}$ definimos $S[i_1, i_2][j_1, j_2] = \sum_{i=i_1}^{i_2} \sum_{j=j_1}^{j_2} A_{ij}$. Con un preproceso de $O(mn)$ podemos dar $S[i_1, i_2][j_1, j_2]$ en tiempo constante.

Sea $S[m, n] = \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij}$. Tenemos

$$S[m][n] = S[m-1][n] + S[m][n-1] - S[m-1][n-1],$$

$S[0][X] = 0$, $S[X][0] = 0$, lo que nos permite calcularlos en $O(mn)$. Ahora

$$S[i_1, i_2][j_1, j_2] = S[i_2][j_2] - S[i_2][j_1] - S[i_1][j_2] + S[i_1][j_1].$$

Maximum subsum. Dados $a_1, \ldots, a_n \in \mathbb{R}$ encontrar $p, q \in [n]$ con $\sum_{i=p}^{q} a_i$ máximo.

Sea

$$dp[q] = \max \left\{ \sum_{i=p}^{q} a_q \;\middle|\; p \leqq q \right\}.$$

Tenemos $dp[q] = a_q + \max\{dp[q-1], 0\}$ y $dp[0] = 0$. La respuesta es $\max_{q \in [n]} dp[q]$. Tiempo $O(n)$.

0/1 Knapsack. Dados $b_1, \ldots, b_n \in \mathbb{N}$, $w_1, \ldots, w_n \in \mathbb{N}$ y $M \in \mathbb{N}$ encontrar $I \subset [n]$ tal que $\sum_{i \in I} b_i$ sea máximo bajo la restricción $\sum_{i \in I} w_i \leqq M$.

Sea

$$dp[m][k] = \min \left\{ \sum_{i \in I} b_i \;\middle|\; I \subset [m], \sum_{i \in I} w_i \leqq k \right\}.$$

Entonces

$$dp[m][k] = \min\{dp[m-1][k - w_m] + b_m, dp[m-1][k]\},$$

con $dp[0][k] = 0$. Esto da un algoritmo de tiempo $O(nM)$.

Subset sum. Dados $a_1, \ldots, a_n \in \mathbb{N}$ y $M \in \mathbb{N}$ determinar si hay $I \subset [n]$ con $\sum_{i \in I} a_i = M$.

Sea

$$dp[m][k] = (\exists I \subset [m]) \left(\sum_{i \in I} a_i = k \right).$$

Entonces

$$dp[m][k] = dp[m-1][k - a_m] \vee dp[m-1][k],$$

con $dp[0][X] = (X = 0)$. Esto da un algoritmo de tiempo $O(nM)$.

Longest common subsequence. Dados $a_1, \ldots, a_N$, $b_1, \ldots, b_M$ encontrar $i_1 < \cdots < i_t$ en $[N]$, $j_1 < \cdots < j_t$ en $[M]$ con $a_{i_k} = b_{j_k}$ para todo $k \in [t]$ y t máximo.

Si $dp[n][m]$ es el problema sobre $a_1, \ldots, a_n$ y $b_1, \ldots, b_m$ entonces tenemos

$$dp[n][m] = \max\{dp[n-1][m-1] + 1, dp[n-1][m], dp[n][m-1]\}$$

si $a_n = b_m$ y $dp[n][m] = \max\{dp[n-1][m], dp[n][m-1]\}$ si no, con $dp[0][X] = 0, dp[X][0] = 0$. Tiempo $O(NM)$.

Longest increasing subsequence. Dados $a_1, \ldots, a_N \in \mathbb{R}$ encontrar $i_1 < \cdots < i_t$ en $[N]$ con $a_{i_1} \leqq \cdots \leqq a_{i_t}$ y t máximo.

Si $dp[n]$ es el problema sobre $a_1, \ldots, a_n$,

$$dp[n] = \max\{dp[k] \mid 0 \leqq k < n, a_k \leqq a_n\} + 1,$$

con $dp[0] = 0$, $a_0 = -\infty$, que se calcula en $O(N^2)$. Se puede mejorar ya que para cada valor de $dp[k]$ solo hay que mirar el que minimiza a_k. Si en el paso n tenemos $t = \max\{dp[k] \mid 0 \leqq k < n\}$, guardamos k_i con $dp[k_i] = i$, a_{k_i} mínimo, para $i = 0, \ldots, t$. Esta secuencia cumple que $a_{k_0} \leqq \cdots \leqq a_{k_t}$ así que con búsqueda binaria localizamos k_i con i máximo y $a_{k_i} \leqq a_n$. Se tiene $a_{k_i} > a_{k_{i+1}}$ (si $i + 1 \leqq t$) así que k_{i+1} va a pasar a ser n. El tiempo total es $O(N \log t)$.

Bellman-Ford. Dado un grafo pesado y un vértice u determinar la distancia a los demás.

Sea $dp[k][v]$ la longitud del camino mínimo de u a v usando a lo sumo k aristas. Tenemos

$$dp[k][v] = \min\{dp[k-1][v], \min_{wv \in E}\{dp[k-1][w] + p[wv]\}\},$$

con $dp[0][v] = \infty$, $dp[0][u] = 0$. La respuesta es $dp[n-1][v]$ si el grafo no tiene ciclos negativos, ya que siempre va a haber un camino mínimo simple (si tiene ciclos son no negativos y los sacamos), así que el tiempo es $O(nm)$. Se ve que el grafo tiene un ciclo negativo si y sólo si en el paso n alguna distancia decrece.

Floyd. Dado un grafo pesado dar la distancia entre cualesquiera dos vértices.

Sea $dp[k][u][v]$ la longitud del camino mínimo de u a v usando exclusivamente los primeros k vértices. Tenemos

$$dp[k][u][v] = \text{mín}\{dp[k-1][u][v], dp[k-1][u][k] + dp[k-1][k][v]\},$$

con $dp[0][u][v]$ es cero si $u = v$, $p[uv]$ si $uv \in E$ y ∞ si no. La respuesta será $dp[n][u][v]$ así que el tiempo es $O(n^3)$. Se ve que el grafo tiene un ciclo negativo si y sólo si hay u con $d[n][u][u] \neq 0$.

Camino en DAG. Dado un grafo dirigido sin ciclos pesado encontrar el camino más largo.

Ordenamos los vértices topológicamente. Sea $dp[k]$ el problema mirando hasta el vértice de orden k. Tenemos

$$dp[k] = \text{máx}\{dp[k-1], \max_{jk \in E}\{dp[j] + p[j][k]\}\},$$

con $dp[0] = 0$. La respuesta es $dp[n]$ y sale en tiempo lineal.

Maximum weighted independent set en árboles. Dado un árbol con una función $w : V \to \mathbb{R}$ encontrar un independent set de peso máximo.

Tomamos un vértice u y orientamos las aristas para que u sea la raíz. Ordenamos los vértices por distancia a la raíz de mayor a menor. Sea $dp_1[k]$ el maximum independent set del subárbol que "cuelga" del vértice de orden k, conteniendo a k; sea $dp_2[k]$ lo mismo pero sin incluir a k. Tenemos

$$dp_1[k] = \text{máx}\left\{\sum_{jk \in E} dp_1[k], w(k) + \sum_{jk \in E} dp_2[k]\right\}, dp_2[k] = \sum_{jk \in E} dp_1[j].$$

La respuesta es $dp_1[n]$ y sale en tiempo lineal.

Maximum independent set en cactus. Un grafo es *cactus* si toda arista está en a lo sumo un ciclo. Encontrar un maximum independent set.

Identificamos los ciclos y armamos un árbol que es el inicial con los ciclos comprimidos a un vértice de manera que si dos ciclos están unidos por un vértice separamos ese vértice y lo unimos por una arista "especial". Definimos un orden de los vértices en el nuevo árbol como en el problema anterior. Usando esa recursión el problema se reduce a, dado un ciclo C, un vértice u marcado, y dos números por cada vértice restante que dicen el maximum independent set del subgrafo que "cuelga" conteniendo o no al vértice, dar un maximum independent set total conteniendo a u y otro no conteniendo a u. Pesamos los vértices que no son u con la diferencia entre meter ese vértice en un independent set y no meterlo. El primer maximum independent set tendrá cardinal uno más el weighted maximum independent set del camino $C - u - N(u)$. El segundo será el del camino $C - u$. Sabemos resolver los dos por lo anterior así que ya está. El tiempo total es lineal.

Matrix chain multiplication. Se tienen N matrices $A_0, \ldots, A_{n-1}$, con $A_i \in \mathbb{R}^{a_i \times a_{i+1}}$. Multiplicar $A \in \mathbb{R}^{a \times b}$ por $B \in \mathbb{R}^{b \times c}$ cuesta abc. Encontrar la manera de poner paréntesis en la multiplicación de manera que multiplicar todas en ese orden resulte lo menos costoso posible.

Sea $dp[i][k]$ con $i < k$ el problema sobre $A_i, \ldots, A_{k-1}$. Entonces

$$dp[i][k] = \min_{i<j<k}\{dp[i][j] + dp[j][k] + a_i a_j a_k\},$$

$dp[i][i+1] = 0$. La respuesta es $dp[0][n]$ y sale en $O(n^3)$.

Hirschberg's divide-and-conquer trick. Queremos encontrar explícitamente una subsecuencia común de longitud máxima entre dos secuencias $a_1, \ldots, a_n$ y $b_1, \ldots, b_m$ en tiempo $O(nm)$ pero con memoria $O(n+m)$. La recursión sólo usa $dp[n-1][m-1]$, $dp[n-1][m]$ y $dp[n][m-1]$, así con $O(n+m)$ memoria se consigue la respuesta. Una LCS entre (a_i) y (b_i) es la concatenación de una LCS entre $a_1, \ldots, a_{n/2}$ y $b_1, \ldots, b_h$, y $a_{n/2+1}, \ldots, a_n$ y $b_{h+1}, \ldots, b_m$. Hacemos que $h[i][j]$, para $i > n/2$, sea un h válido suponiendo que el problema es sobre $a_1, \ldots, a_i$ y $b_1, \ldots, b_j$. Entonces $h[i][j]$ es $h[i-1][j]$ si $dp[i][j] = dp[i-1][j]$, $h[i][j-1]$ si $dp[i][j] = dp[i][j-1]$, $h[i-1][j-1]$ si $dp[i][j] = dp[i-1][j-1]+1$, con $h[n/2][j] = j$. Podemos calcular $h[n][m]$ junto con dp en tiempo y memoria. Ahora el problema se reduce a calcular dos LCS explícitas de tamaños $n/2, h$ y $n/2, m-h$. Lo hacemos recursivamente. El tiempo cumple $T(m,n) \leqq O(nm) + T(n/2,h) + T(n/2, m-h)$, que se ve que es $O(nm)$.

Sparseness. Dadas dos secuencias $a_1, \ldots, a_n$ y $b_1, \ldots, b_m$, vamos a resolver LCS en tiempo $O(n \log n + m \log m + K \log L)$, donde K es la cantidad de pares $(i,j) \in [n] \times [m]$ tales que $a_i = b_j$ y L es la respuesta. Primero las ordenamos e imitando una operación merge encontramos el conjunto M de todos esos pares. Es claro que buscamos máx$\{dp[i][j] \mid (i,j) \in M\}$. Ahora $dp[i][j]$ es el máximo de $1 + dp$ sobre los (i',j') con $i' < i$ y $j' < j$ y dp sobre los (i,j') con $j' < j$ y sobre los (i',j) con $i' < i$. Entonces lo que buscamos es una LIS sobre M con el orden $(i_1,j_1) < (i_2,j_2)$ si y sólo si $i_1 < i_2$ y $j_1 < j_2$. Ordenamos por i y es LIS sobre los j, que sale en $O(K \log L)$. Sumando lo que costó ordenar y hacer el merge da el tiempo propuesto.

1.2.3 Greedy method

La idea de un algoritmo greedy es tomar decisiones teniendo en cuenta sólo información local sobre el problema. Es fácil crear algoritmos greedy para un problema, pero la gran mayoría de las veces son malos. Sin embargo hay casos en los que el método funciona; los algoritmos resultantes suelen ser simples y eficientes.

Reordenamiento. Tenemos dos secuencias de números reales $a_1 > a_2 > \ldots > a_n$ y $b_1, \ldots, b_n$ (la primera ordenada). Queremos el máximo de $a_1 b_{\sigma_1} + \cdots + a_n b_{\sigma_n}$ sobre las permutaciones $\sigma \in S_n$. El algoritmo greedy sería: poner al mayor de la primera secuencia con el mayor de la segunda, y seguir con el resto. Esto funciona: tomemos σ máxima; si hay $i < j$ con $b_{\sigma_i} < b_{\sigma_j}$ los cambiamos y la diferencia es $a_i b_{\sigma_j} + a_j b_{\sigma_i} - a_i b_{\sigma_i} - a_j b_{\sigma_j} = (a_i - a_j)(b_{\sigma_j} - b_{\sigma_i}) > 0$, absurdo.

Optimal caching. Supongamos que tenemos una secuencia $d_1, \ldots, d_n \in [m]$ y un conjunto $M \subset [m]$ de k elementos. Se recorre la secuencia: en el paso i, si $d_i \notin M$ se toma alguno de M y se lo cambia por d_i. El problema es determinar en cada paso por cuál elemento cambiar de manera que la cantidad de cambios total sea mínima.

El algoritmo es: si hay que cambiar, sacar de M el elemento que aparece más adelante en la secuencia. La prueba: sea S^* la secuencia de conjuntos M que genera el algoritmo, sea S otra; si coinciden en los primeros j conjuntos, vemos que hay S' que coincide con S^* en los primeros $j+1$ y no es peor que S. Supongamos que en el paso $j+1$ el más lejano de M es f pero S descarta e; de acá en adelante hacemos que S' haga lo que hace S hasta que haya un d que no es e ni f tal que S lo mete eliminando e, en cuyo caso lo metemos eliminando f y a partir de ahí seguimos como S, o hasta que haya un d que sea e, en cuyo caso si S elimina f quedan iguales, y si S elimina a otro, e', S pasa de tener f, e' a tener f, e así que podemos hacer que S' pase de e, e' a e, f, pero esto recién cuando un d sea f, y mientras tanto y después podemos imitar a S.

Códigos de Huffman. Sea Σ un alfabeto y $\{0,1\}^*$ el conjunto de todas las strings formadas por ceros y unos. Un código binario es una asignación $c : \Sigma \to \{0,1\}^*$; es libre de prefijos

si para todo $\sigma_1, \sigma_2 \in \Sigma$ con $\sigma_1 \neq \sigma_2$ se cumple que $f(\sigma_1)$ no es prefijo de $f(\sigma_2)$. Se puede ver como un árbol binario en el que las hojas son los elementos de Σ y el camino de la raíz a cada letra representa su código asignado: el camino es una secuencia de direcciones izquierda o derecha, que representan el cero y el uno, respectivamente. La pregunta es, dado un alfabeto Σ y frecuencias $f : \Sigma \to \mathbb{N}$, encontrar un código libre de prefijos $c : \Sigma \to \{0,1\}^*$ tal que la suma $\sum_{\sigma\in\Sigma} f(\sigma)|c(\sigma)|$ sea mínima. El algoritmo es: tomamos las dos letras con frecuencia menor σ_1, σ_2, las eliminamos, agregamos una letra ficticia σ' cuya frecuencia es $f(\sigma_1) + f(\sigma_2)$, seguimos recursivamente, y al final los códigos para σ_1 y σ_2 son el de σ' con un cero agregado al final, y con un uno al final, respectivamente. Se implementa con una binary heap en tiempo $O(n \log n)$, donde $n = |\Sigma|$. La demostración de correctitud es fácil.

Min-cost arborescence. Dado un grafo dirigido G con pesos $w : E \to \mathbb{R}_0^+$ y $r \in V$ tal que para todo $v \in V$ se tiene $r \rightsquigarrow v$ se busca un subgrafo G' con $V(G') = V(G)$ que sea un árbol con raíz en r y tal que la suma $\sum_{e\in E(G')} w(e)$ sea mínima.

El algoritmo es: tomamos las aristas uv y ponemos $w'(uv) = w(uv) - \min_{u'v\in E} w(u'v)$ (siguen siendo no negativas, y como toda arborescence contiene exactamente una arista incidente a cada vértice salvo r, esto mantiene el óptimo); si las aristas con peso cero forman una arborescence, es óptima y terminamos; si no, debe haber un ciclo, así que lo contraemos a un vértice y resolvemos recursivamente el nuevo problema; aumentamos la arborescence que obtenemos agregando todas las aristas del ciclo menos una. Hay que probar que si C es un ciclo de costo cero entonces hay una min-cost arborescence tal que le entra exactamente un arista; tomamos una óptima; tomamos un camino mínimo de r a C, ponemos la arista e que entra a C, sacamos el resto de las aristas que entran a C y ponemos todas las aristas de C salvo la que no va con e; lo obtenido no es peor que el óptimo así que basta ver que es una arborescence: se ve que a cada vértice entra exactamente una arista así que resta ver que es conexo y a su vez esto se reduce a ver que hay un camino de r a e, pero esto es claro por la condición de minimalidad de e.

1.2.4 Ejercicios

1. Dadas dos listas de números ordenados de tamaños m y n y un número k, encontrar el k-ésimo menor elemento en la unión de las dos listas en tiempo $O(\log m + \log n)$. Lo mismo con tres de longitud n en tiempo $O(\log n)$, y con m listas de n elementos en $O(m \log n)$.

2. (Inversiones) Dados números $a_1, \ldots, a_n$ contar la cantidad de pares $i, j \in [n]$ con $i < j$ y $a_i > a_j$ en $O(n \log n)$.[12]

3. (Closest pair) Dados puntos $a_1, \ldots, a_n \in \mathbb{R}^2$ encontrar el par a_i, a_j con $i \neq j$ a distancia mínima en $O(n \log n)$.

4. (Kd-tree) Tenemos n puntos $a_1, \ldots, a_n \in \mathbb{R}^2$ en un rectángulo con lados paralelos a los ejes. Un *kd-tree* recursivamente subdivide los puntos: primero divide la caja en dos con una línea vertical que pasa por el punto que es la mediana en el orden por la coordenada x (si hay dos medianas toma cualquiera); luego, en cada una de las dos cajas subdivide con líneas horizontales que pasan por los puntos que son la mediana según el orden por la coordenada y; así sigue alternando divisiones verticales y horizontales. Para cuando no quedan cajas con puntos en su interior. Usando esta idea, con un preproceso de $O(n \log n)$, decir cuántos puntos hay debajo de una línea horizontal o vertical y cuántos puntos hay dentro de un rectángulo dado con lados paralelos a los ejes en tiempo $O(\sqrt{n})$.

[12]Sugerencia. Modificar merge sort.

5. (Seidel) Dado un grafo G no dirigido, sin pesos, conexo, representado por la matriz de adyacencia $A \in \{0,1\}^{n\times n}$, calcular la matriz $D \in \mathbb{N}_0^{n\times n}$ en la cual D_{ij} es la longitud del camino más corto del vértice i al j en tiempo $O(n^{\log_2 7}\log n)$.[13]

6. (Convolución) Si $a_0,\ldots,a_n$ y $b_1,\ldots,b_m$ son números complejos, hallar

$$\sum_{i+j=0} a_i b_j,\ \sum_{i+j=1} a_i b_j,\ \ldots,\ \sum_{i+j=n+m} a_i b_j$$

en tiempo $O((n+m)\log(n+m))$.

7. (Chirp transform) Si $a_0,\ldots,a_n, z \in \mathbb{C}$, hallar $y_0,\ldots,y_n$, donde $y_k = \sum_{j=0}^n a_j z^{kj}$ en tiempo $O(n\log n)$.[14]

8. Si aplicamos FFT sobre el anillo $\mathbb{Z}_m$ de los números módulo $m = 2^{tn/2}+1$, donde t es un entero positivo arbitrario, podemos usar $\omega = 2^t$.

9. Dados dos conjuntos X, Y de números enteros calcular $X+Y = \{x+y \mid x\in X, y\in Y\}$ en tiempo $O(M\log M)$, donde M es $\max\limits_{x\in X\cup Y}|x|$.

10. (Jeff Erickson) Dados números $a_1 < a_2 < \cdots < a_n$ encontrar la longitud de la progresión aritmética más larga contenida en esos números, es decir, el número k máximo tal que hay d y $1 \leqq i_1 < \cdots < i_k \leqq n$ con $a_{i_{j+1}} - a_{i_j} = d$ para $j = 1,\ldots,n-1$ en tiempo $O(n^2)$ y $O((n^2/k)\log(n/k)\log k)$.

11. Dada $A \in \mathbb{R}^{n\times m}$ encontrar $i_1, i_2 \in [n], j_1, j_2 \in [m]$ con $\sum_{i=i_1}^{i_2}\sum_{j=j_1}^{j_2} A_{ij}$ máximo en tiempo $O(nm\,\text{mín}\{n,m\})$.

12. (Scheduling Classes) Dados reales $s_1,\ldots,s_n$, $f_1,\ldots,f_n$ y $w_1,\ldots,w_n$ con $s_i < f_i$ para todo $i\in[n]$, encontrar $I\subset[n]$ tal que si $i,j\in I$ con $s_i < s_j$ entonces $f_i < s_j$ y tal que si $s_i = s_j$ entonces $i = j$, con $\sum_{i\in I} w_i$ máximo en $O(n\log n)$.

13. (Calcetines) Dados $b_1,\ldots,b_n \in \mathbb{N}$ y $a_1,\ldots,a_n \in \mathbb{Z}_2^p$ encontrar $I\subset[n]$ con $\sum_{i\in I} b_i$ máxima dado $\sum_{i\in I} a_i = 0$ en tiempo $O(n2^p)$.

14. (Yogures) Hay k tipos de yogures, con precios $p_1,\ldots,p_k$, con kalorías $w_1,\ldots,w_k$. Hay que comer yogur durante n días de manera que no se use el mismo yogur dos días consecutivos y la cantidad total de calorías sea a lo sumo K. Elegir los yogures minimizando el precio en tiempo $O(nk^2K)$.

15. (DNA Sequences) Dados $a_1,\ldots,a_m$, $b_1,\ldots,b_n$, k se quiere encontrar la subsecuencia común más larga de (a_i) y (b_i) formada por segmentos de longitud k o mayor, en tiempo $O(mn)$.

16. (Box nesting) Si $a,b\in\mathbb{R}^m$ decimos que $a \leqq b$ si hay una permutación $\pi\in S_m$ con $a_1 \leqq b_{\pi_1},\ldots,a_m \leqq b_{\pi_m}$. Dada una secuencia de n elementos de $\mathbb{R}^m$ dar la longest increasing subsequence con ese orden en tiempo $O(nm(n+\log m))$.

17. (Arbolito) Hay una senda de longitud L donde se plantan en fila n árboles. La distancia del i-ésimo al origen es a_i. Se tiene que sus alturas son crecientes. Se quiere replantar algunos, la mínima cantidad posible, de manera que la distancia entre dos consecutivos sea al menos d y las alturas sigan estando en orden creciente. Determinar esa mínima cantidad en tiempo $O(n\log n)$.

---

[13] Sugerencia. Sea G^2 el grafo con los vértices de G tal que $uv\in E(G^2)$ si y sólo si $D[u,v] \leqq 2$. Se calcula la matriz de adyacencia de G^2 en tiempo $O(n^{\log_2 7})$. Si G^2 es completo sale en $O(n^2)$; si no, $D[i,j]$ es $2D^2[i,j]-1$ ó $2D^2[i,j]$, dependiendo si $X[i,j] = \sum_{ik\in E(G)} D^2[i,j] < D^2[i,j]\deg_G(i)$ o no, donde $X = D^2\cdot A$.

[14] Sugerencia. Usar la ecuación $y_k = z^{k^2/2}\sum_{j=0}^n (a_j z^{j^2/2})(z^{-(k-j)^2/2})$.

18. (Johnson) Sea G un grafo con pesos p. Encontrar la distancia entre todo par de vértices en tiempo $O(nm + n^2 \log n)$.[15]

19. Resolver el problema lineal

$$\begin{array}{rrcl} \text{maximizar} & x_i - x_j & & \\ \text{dado} & x_{i_1} - x_{j_1} & \leqq & a_{i_1 j_1} \\ & & \vdots & \\ & x_{i_m} - x_{j_m} & \leqq & a_{i_m j_m} \\ & x_1, \ldots, x_n & \in & \mathbb{R} \end{array}$$

en tiempo $O(nm)$.

20. (Minimum path cover en árbol y en cactus) Dado un árbol partir los vértices en la menor cantidad de caminos en tiempo lineal. Lo mismo para un cactus.

21. (Smallest dominating set en árbol y en cactus) Dado un árbol encontrar un conjunto $S \subset V$ con $S \cup N(S) = V$ y $|S|$ mínimo, en tiempo lineal. Lo mismo para un cactus.

22. (Diámetro en árbol y cactus) El diámetro de un grafo es la máxima distancia mínima entre dos vértices. Dado un árbol con pesos en las aristas encontrar el diámetro en tiempo lineal. Lo mismo para cactus.

23. (Heap-ordered subset) Dado un árbol T con raíz con n nodos y una función $f : V(T) \to \mathbb{R}$ encontrar el mayor subconjunto $S \subset V(T)$ tal que S contiene un nodo que es ancestro de todos los demás nodos de S y tal que si $u, v \in S$ con u ancestro de v entonces $f(u) \leqq f(v)$, en tiempo $O(n \log n)$.[16]

24. (Post Office) Dados $a_1 < a_2 < \ldots < a_n \in \mathbb{N}$ y $m \in [n]$, si $I \in [n]$ definimos $d(i, I)$ como $\min_{j \in I} |a_i - a_j|$. Encontrar I con $|I| = m$ que minimice $\sum_{i=1}^{n} d(i, I)$ en tiempo $O(n^2 m)$.

25. Dados $a_1 < a_2 < \ldots < a_n \in \mathbb{N}$ y $d \in \mathbb{N}$ encontrar $I \in [n]$ mínimo con $\max_{i \in [n]} \min_{j \in I} |a_i - a_j| \leqq d$ en tiempo $O(n)$.[17]

26. Dados $a_1 < a_2 < \ldots < a_n \in \mathbb{N}$ y $d \in \mathbb{N}$ encontrar $I \in [n]$ mínimo con $\max_{i \in [n]} \min_{j \in I} \min\{|a_i - a_j|, a_n - a_1 - |a_i - a_j|\} \leqq d$ en tiempo $O(n)$.

27. (Alarma) Dados $a_1 < a_2 < \ldots < a_n \in \mathbb{N}$ y $m \in [n]$, si $I \subset [n]$ definimos $d(i, I)$ como $\min_{j \in I} \min\{|a_i - a_j|, a_n - a_1 - |a_i - a_j|\}$. Encontrar I con $|I| = m$ que minimice $\max_{1 \leqq i \leqq n} d(i, I)$ en tiempo $O(nm)$ y en tiempo $O(n \log a_n)$.[18]

28. (Storing files on tape) Dados números reales $a_1, \ldots, a_n$ y $b_1, \ldots, b_n$ encontrar $\sigma \in S_n$ tal que

$$\sum_{k=1}^{n} \sum_{i=1}^{k} a_{\sigma_k} b_{\sigma_i}$$

[15] Sugerencia. Agregamos un vértice s a V y aristas de s a todos los demás con peso cero. Largar Bellman-Ford desde s y luego Dijkstra sobre pesos p' dados por $p'(uv) = p(uv) + d(u) - d(v)$.

[16] Sugerencia. Primero considerar el caso en el que el árbol es un camino. Para el caso general hay que hacer merge de listas ordenadas rápido. Representando los nodos como puntos en $\mathbb{R}^2$ donde en x ponemos el número y en y ponemos el tiempo en el que DFS abandona el nodo no hace falta hacer merge, pero hay que buscar y borrar entre los puntos con y en un intervalo. Con un interval tree 2D se puede hacer esto en $O(\log n)$, con mucho cuidado sobre el algoritmo de borrado: para poder buscar en y y borrar va a haber que mantener un 2-4 tree o algo así; para los otros niveles basta con listas. De otra manera, usando el merge en $O(m \log(n/m))$ de splay trees de tamaños m y n, $m < n$, también sale en $O(n \log n)$ total.

[17] Sugerencia. No usar programación dinámica.

[18] Sugerencia. Para conseguir el tiempo $O(nm)$ usar programación dinámica. Para conseguir el tiempo $O(n \log a_n)$ reducir al problema anterior usando búsqueda binaria.

sea mínimo en tiempo $O(n \log n)$.

29. (Least cost bracket sequence) Dado n par y números $a_1, \ldots, a_n$, $b_1, \ldots, b_n$, encontrar una secuencia de paréntesis $p_1 \ldots p_n$ válida, esto es, tal que en $p_1 \ldots p_i$ no haya más paréntesis cerrados que abiertos para $i = 1, \ldots, n$, que minimice $\sum_{p_i \text{ abre}} a_i + \sum_{p_i \text{ cierra}} b_i$ en tiempo $O(n \log n)$.

30. (Interval graphs) Dado un conjunto X de intervalos reales cerrados (es decir, pares (a, b) que representan $[a, b]$), construímos un grafo cuyos vértices son los intervalos y dos intervalos están unidos por una arista si y sólo si tienen intersección no vacía. Encontrar un maximum independent set, maximum weighted independent set, maximum clique y el número cromático en tiempo $O(n \log n)$. Encontrar un conjunto $Y \subset X$ con $|Y|$ mínimo tal que $\bigcup Y = \bigcup X$, y un conjunto $A \in \mathbb{R}$ con $|A|$ mínimo tal que $\bigvee_{x \in X} \bigwedge_{a \in A} a \in x$ en $O(n \log n)$.

31. (Circular arc graphs) Dado un conjunto X de arcos del círculo unitario (es decir, pares $(a, b) \in [0, 2\pi)^2$ que representan $\{e^{i\phi} \mid a \leqq \phi \leqq b\}$), construímos un grafo cuyos vértices son los arcos y dos arcos están unidos por una arista si y sólo si tienen intersección no vacía. Encontrar un maximum independent set, maximum clique y el número cromático en tiempo $O(n \log n)$. Encontrar un conjunto $Y \subset X$ con $|Y|$ mínimo tal que $\bigcup Y = \bigcup X$, y un conjunto $A \in [0, 2\pi)$ con $|A|$ mínimo tal que $\bigvee_{x \in X} \bigwedge_{\phi \in A} e^{i\phi} \in x$ en $O(n \log n)$.

32. (Job scheduling) Dados n trabajos que duran $l_1, \ldots, l_n$, resp., y deadlines $d_1, \ldots, d_n$, resp., encontrar $\sigma \in S_n$ tal que $\max\limits_{1 \leqq i \leqq n} \max\{l_{\sigma_1} + \cdots + l_{\sigma_i} - d_{\sigma_i}, 0\}$ sea mínimo en tiempo $O(n \log n)$.

1.3 Flujos y matchings

1.3.1 Flujos

Una red de flujo es una tupla $N = (G, c, s, t)$, donde G es un digrafo (V, E), c es una capacidad $c : V^2 \to \mathbb{R}_0^+$, donde $c(u, v) = 0$ si $(u, v) \notin E$, y $s, t \in V$ de manera que si $v \in V$, $s \rightsquigarrow v \rightsquigarrow t$. Un flujo sobre la red es una función $f : V^2 \to \mathbb{R}$ que cumple los siguientes axiomas:

F1 Si $u, v \in V$, $f(u, v) \leqq c(u, v)$.
F2 Si $u, v \in V$, $f(u, v) = -f(v, u)$.
F3 Si $u \in V \setminus \{s, t\}$ entonces $\sum_{v \in V} f(u, v) = 0$.

El valor de un flujo se define como $|f| = \sum_{v \in V} f(s, v)$. El problema del flujo máximo es dada una red encontrar un flujo de valor máximo. El flujo positivo total que entra y que sale a un vértice se define como

$$f_i^+(v) = \sum_{\substack{u \in V \\ f(u,v) > 0}} f(u, v) \quad \text{y} \quad f_o^+(v) = \sum_{\substack{u \in V \\ f(v,u) > 0}} f(v, u).$$

Ahora **F3** es equivalente a que si $v \in V \setminus \{s, t\}$, $f_i^+(v) = f_o^+(v)$. Escribimos $f(X, Y) = \sum_{u \in X} \sum_{v \in Y} f(u, v)$. Tenemos lo siguiente:

L1 Si $X \subset V$ entonces $f(X, X) = 0$.
L2 Si $X, Y \subset V$ entonces $f(X, Y) = -f(Y, X)$.
L3 Si $X, Y, Z \subset V$ con $X \cap Y = \emptyset$ entonces $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$.
L4 $|f| = f(s, V) = f(V, t)$.

Si N es una red y f es un $N-$flujo definimos la red residual inducida por f como $N_f = (V, E_f, c_f, s, t)$, donde $c_f = c - f$ y $E_f = \{(u, v) \in V^2 \mid c_f(u, v) > 0\}$. Un camino de aumento p es un camino de s a t en la red residual N_f. La capacidad residual de p se define como $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$.

L5 Si N es una red, f es un $N-$flujo y f' es un N_f-flujo entonces $f+f'$ es un $N-$flujo con $|f+f'| = |f| + |f'|$.

L6 Si N es una red, f es un $N-$flujo y p es un f-camino de aumento y $f_p : V^2 \to \mathbb{R}$ definida por

$$f_p(u,v) = \begin{cases} c_f(p) & \text{si } (u,v) \text{ está en } p, \\ -c_f(p) & \text{si } (v,u) \text{ está en } p, \\ 0 & \text{en otro caso} \end{cases}$$

es un flujo en G_f con $|f_p| = c_f(p) > 0$.

Un corte de una red es una partición (S,T) de V donde $s \in S$ y $t \in T$. Si f es un flujo, definimos el flujo del corte (S,T) como $f(S,T)$ y la capacidad como $c(S,T)$. Un corte mínimo es aquel cuya capacidad es mínima.

L7 Si f es un $N-$flujo y (S,T) es un $N-$corte entonces $|f| = f(S,T)$.

L8 Luego $|f| \leqq c(S,T)$.

Se tiene el siguiente

Teorema (Max-flow min-cut) *Si $N = (V,E,c,s,t)$ es una red y f es un $N-$flujo entonces son equivalentes:*

(1) *f es un flujo máximo,*
(2) *la red N_f no tiene caminos de aumento y*
(3) *$|f| = c(S,T)$ para algún corte (S,T) de N.*

Prueba En efecto, (1) $\Rightarrow$ (2) es obvio. Sea $S = \{v \mid s \rightsquigarrow v \text{ en } G_f\}$ y $T = V - S$. Ahora (S,T) es un corte y $f(u,v) = c(u,v)$ si $u \in S, v \in T$, ya que $c_f(u,v)$ es cero. Entonces $|f| = f(S,T) = c(S,T)$ y tenemos (2) $\Rightarrow$ (3). Ahora (3) $\Rightarrow$ (1) es obvio y terminamos. ∎

Entonces uno puede construir flujos máximos con el siguiente método: Se empieza con $f = 0$. En cada paso, se ve la red residual y se busca un camino de aumento. El flujo inducido por el camino se suma a f. Se sigue mientras se encuentren caminos de aumento. Este es el método de Ford-Fulkerson. Si el problema es entero, en cada iteración aumenta el flujo en al menos uno, por lo que son a lo sumo f iteraciones.

Corolario El máximo flujo es igual a la mínima suma de capacidades de un conjunto de aristas que desconectan s y t.

En efecto, si D es un tal conjunto, hacemos que S sea $\{v \mid s \rightsquigarrow v \text{ en } (V, E \setminus D)\}$. Toda arista $uv \in E$ con $u \in S$, $v \in V \setminus S$ tiene que estar en D así que $\sum_{uv \in D} c(u,v) \geqq c(S, V \setminus S)$. Por otro lado si (S^*, T^*) es un corte mínimo tenemos

$$c(S, V \setminus S) \geqq c(S^*, T^*) = \sum_{\substack{uv \in E \\ u \in S^*, v \in T^*}} c(u,v) \geqq \sum_{uv \in D} c(u,v),$$

la última desigualdad porque $D^* = \{uv \in E \mid u \in S^*, v \in T^*\}$ es un conjunto de aristas que desconecta s y t y D es mínimo. Entonces son todas igualdades y podemos aplicar max-flow min-cut.

Sea $G = (V,E)$ un grafo bipartito con $V = (A,B)$. Lo extendemos a una red: le agregamos dos vértices, s y t, de manera que si $u \in A$ entonces $(s,u) \in E$, si $u \in B$ entonces $(u,t) \in E$ y si $uv \in E$ con $u \in A, v \in B$ entonces queda dirigido (u,v). A todas las aristas les ponemos capacidad uno. Vemos que un matching máximo en el grafo bipartito se corresponde con un flujo máximo en la red. Es cuestión de largar Ford-Fulkerson, lo cual da un flujo entero, ver

que induce un matching y que un matching induce un flujo. Entonces el matching máximo es el mínimo corte.

Un *vertex covering* del grafo es un conjunto de vértices C tal que $V \setminus C$ es independiente, es decir, que C "cubre" todas las aristas. Veamos cómo se puede relacionar con el máximo flujo.

Teorema (Kőnig) *Si G es un grafo bipartito de n vértices cuyo flujo máximo f cumple $|f| = m$ y con $\alpha = \alpha(G)$ se tiene $\alpha = n - m$.*

Prueba Aplicamos max-flow min-cut a la red que asociamos al grafo bipartito. El flujo máximo f es igual a la mínima cantidad de aristas necesaria para desconectar la red. Podemos suponer que estas aristas son incidentes a s o a t. Los otros vértices incidentes a estas aristas forman un vertex cover y todo vertex cover induce un corte. Entonces f es igual a la cardinalidad de un mínimo vertex cover, que es $n - \alpha$. ∎

Teorema (Hall) *Si G es un grafo bipartito entonces existe un matching perfecto si y sólo si para todo subconjunto S de A, $|\bigcup_{u \in S} \mathrm{N}(u)| \geqq |S|$.*

Prueba Una implicación es obvia. Para la otra, tomemos un vertex cover mínimo C. Suponemos que $A \setminus C$ es no vacío. Ahora $|\bigcup_{u \in A \setminus C} \mathrm{N}(u)| \subset C \cap B$ así que $|A \setminus C| \leqq |C \cap B|$. Pero $|C| = |C \cap A| + |C \cap B| = |A| - |A \setminus C| + |C \cap B| \geqq |A|$ así que por el teorema de Kőnig tenemos el resultado. ∎

1.3.2 Dinic, Karzanov-Tarjan y Stoer-Wagner

Un blocking flow es un flujo tal que todo camino mínimo de s a t tiene una arista saturada, es decir, que en la red residual no va a estar, por lo que en ésta la distancia de s a t va a ser necesariamente mayor. (Ninguna distancia decrece tomando red residual de un flujo aumentado solo en caminos mínimos, ya que justamente el efecto que produce es romper estos caminos.) La distancia de s a t puede aumentar solo $n - 1$ veces: de 1 a 2, de 2 a 3, y así siguiendo hasta pasar de $n - 1$ a infinito, en cuyo caso no hay más caminos de aumento y el flujo acumulado es máximo.

Encontrar el blocking flow se puede hacer de una manera sencilla: miramos el grafo de nivel, que es el árbol del BFS. Vamos avanzando desde s hasta que no podemos seguir. Si llegamos a t eliminamos las aristas saturadas y aumentamos el flujo. Si no llegamos, borramos la última arista del camino, volvemos atrás y seguimos avanzando, que sería como hacer backtracking en DFS. Avanzar desde s toma tiempo $O(k)$ hasta que no podems seguir, donde k es la cantidad de niveles. En cada avance borramos al menos una arista así que como mucho son m avances, lo cual da $O(mk)$. Como $k \leqq n$, esto es $O(mn)$. El BFS toma tiempo $O(m)$ que es despreciable. Son a lo sumo $n - 1$ blocking flows, así que en total el tiempo es $O(n^2 m)$. Este es el algoritmo de Dinic.

Teorema *El algoritmo de Dinic encuentra el máximo flujo en tiempo $O(n^2 m)$.*

Una red unitaria es una red en la que todas las capacidades son enteras y todo vértice salvo s y t tiene sólo una arista entrando, de capacidad uno, o sólo una arista saliendo, de capacidad uno.

Teorema *El algoritmo de Dinic encuentra el máximo flujo en una red unitaria en tiempo $O(\sqrt{n}m)$.*

Prueba Veamos que sólo busca $O(\sqrt{n})$ blocking flows. Sea f^* un flujo máximo y f un flujo entero. Tenemos que $f^* - f$ es un flujo en la red residual de f, que es cero o uno en cada arista. Podemos partir las aristas en las que $f^* - f$ es uno en una colección de $|f^*| - |f|$ caminos de s a t y posiblemente algunos ciclos. Cada vértice está en a lo sumo un camino, así que por palomar

un camino mide como mucho $(n-2)/(|f^*|-|f|)+1$. Tras $\lceil\sqrt{n-2}\rceil$ blocking flows, el camino más corto tiene longitud al menos $\sqrt{n-1}+2$, así que $\sqrt{n-1}+2 \leqq (n-2)/(|f^*|-|f|)+1$ y $|f^*|-|f| \leqq \sqrt{n-2}$. Así que en a lo sumo $\lfloor\sqrt{n-2}\rfloor$ blocking flows más se va a encontrar el máximo flujo.

Ahora veamos que encontrar un blocking flow toma tiempo $O(m)$. Hacer el BFS toma tiempo $O(m)$. Cada vez que aumentamos el flujo, como la red es unitaria, borramos al menos $(k-1)/2$ aristas, y el tiempo que tardamos es $O(k)$, así que en total aumentar suma como mucho $2m/(k-1)O(k) = O(m)$ más. Cada vez que avanzamos una arista la borramos o aumentamos su flujo así que como mucho avanzamos $O(m)$ veces. Así que el tiempo total es $O(m)$. ∎

En particular notemos que se aplica a la red construida para el matching bipartito, mejorando la complejidad que daba Ford-Fulkerson $O(mn)$ a $O(\sqrt{n}m)$.

Para redes densas el método se puede mejorar. Veamos el *wave method*, una simplificación de Tarjan a un método de Karzanov para encontrar un blocking flow en tiempo $O(n^2)$. La idea también es trabajar sobra el grafo de nivel del BFS, pero en lugar de ir aumentando un flujo se comienza con un *preflujo* y se lo va corrigiendo hasta que se convierte en flujo. Un *preflujo* es una función $f : V^2 \to \mathbb{R}$ tal que $f(u,v) = -f(v,u)$, $f(u,v) \leqq c(u,v)$ y $\Delta f(v) = \sum_{u\in V} f(u,v) \geqq 0$. Decimos que v está balanceado si $\Delta f(v) = 0$. Los vértices están bloqueados o no a lo largo de la ejecución del algoritmo. Inicialmente sólo s está bloqueado y algunos se van desbloqueando. Los vértices no bloqueados se balancean incrementando flujo a sus hijos: si $(v,w) \in E$ no está saturado y w no está bloqueado, incrementamos $f(v,w)$ en $\min\{c(v,w)-f(v,w), \Delta f(v)\}$. Esto puede no terminar de balancear. Balanceamos un vértice v bloqueado decrementando $f(u,v)$ en $\min\{f(u,v), \Delta f(v)\}$ para aristas (u,v). Entonces para encontrar un blocking flow empezamos con el preflujo que satura las aristas que salen de s y es cero en todas las demás y repetimos los siguientes pasos hasta que no haya vértices no balanceados: *Incrementar.* Pasa por los vértices de s a t en orden topológico; si v no está balanceado ni bloqueado se trata de balancearlo; si no se logra se lo bloquea. *Decrementar.* Pasa por los vértices de s a t en orden topológico al revés, balanceando los vértices no balanceados y bloqueados.

Teorema *El wave method correctamente encuentra un blocking flow en tiempo $O(n^2)$.*

Prueba Se mantiene invariante que si v está bloqueado, todo camino de v a t tiene una arista saturada. Como s está bloqueado al principio, todo preflow va a ser blocking. Termina cuando es un flujo, así que es correcto. Como mucho se pasa $n-1$ veces por *Incrementar* y *Decrementar*, ya que *Incrementar* bloquea siempre al menos un vértice, salvo en el último paso. Así que se incrementa o decrementa localmente a lo sumo $(n-1)(n-2)$ veces. En cada vértice guardamos $\Delta f(v)$, si está bloqueado o no, y la última arista de su vecindad visitada, cosa de no mirar las anteriores consideradas. Entonces son $2m + (n-1)(n-2)$ operaciones, que es $O(n^2)$. ∎

Veamos ahora el algoritmo de Stoer-Wagner para obtener un corte mínimo en un grafo no dirigido con pesos no negativos. La idea es que si tenemos un corte que separa dos vértices s y t, entonces el mínimo corte es ese o el mínimo corte en el grafo original pero pegando s y t, ya que un corte menor los va a tener conectados. Entonces con tener un algoritmo que encuentre un mínimo corte que separe dos vértices ya estamos: podemos aplicarlo recursivamente.

El algoritmo consiste en tomar un vértice cualquiera v_1, e ir tomando en el paso k el vértice v_k con $w(A_k, v_k)$ máximo, donde $A_k = \{v_1, \ldots, v_{k-1}\}$. El corte $(\{v_1, \ldots, v_{n-1}\}, \{v_n\})$ va a ser un $v_{n-1} - v_n$ corte mínimo. En efecto, sea C un v_{n-1}, v_n corte y C_k la restricción del corte a A_{k+1}. Decimos que v_k es activo si v_{k-1} y v_k están separados en C. Veamos que para todos estos se tiene $w(A_k, v_k) \leqq w(C_k)$. En particular v_n es activo y obtenemos que nuestro corte es mínimo. Para el primer v_k activo la desigualdad es igualdad. Si es verdad hasta k, sea j el próximo activo. Ahora $w(A_j, v_j) = w(A_k, v_j) + w(A_k - A_j, v_j) \leqq w(A_k, v_k) + w(A_k - A_j, v_j) \leqq w(C_k) + w(A_k - A_j, v_j) \leqq w(C_j)$. La primera desigualdad porque v_k se eligió así. La segunda por

hipótesis. La tercera porque v_j y $A_k - a_j$ están separados por C así que sus aristas contribuyen a C_j pero no a C_k.

Ir encontrando $w(A_k, v_k)$ máximo se puede hacer manteniendo una binary heap. Son n operaciones extraer-máximo y m aumentar-valor. Las dos cuestan $O(\log n)$ así que la complejidad es $O((n+m)\log n)$. Se ejecuta n veces, en cada una de las cuales se reducen dos vértices a uno. Así que el tiempo total es $O(nm \log n)$. Con una Fibonacci heap es $O(nm + n^2 \log n)$. Sin estructuras es $O(n^3)$. Probamos el siguiente

Teorema *El algoritmo de Stoer-Wagner encuentra el mínimo corte en un grafo no dirigido con pesos no negativos en tiempo* $O(nm + n^2 \log n)$.

1.3.3 Flujo de costo mínimo

Supongamos que tenemos una red de flujo y una función $cost : V^2 \to \mathbb{R}$. Podemos hacer que $cost(u,v) = -cost(v,u)$ usando múltiples aristas. Un flujo de costo mínimo es un flujo f tal que $\sum_{f(u,v)>0} cost(u,v) f(u,v)$ es mínimo entre los flujos de valor $|f|$. Definimos el costo de un camino como la suma de los costos de sus aristas.

Teorema *Un flujo f es de costo mínimo si y solo si su red residual no tiene ciclos de costo negativo.*

Prueba Si R tiene un ciclo de costo negativo podemos reducir el costo de f sin cambiar su valor mandando flujo por ese ciclo. Si f no es de costo mínimo, sea f^* un flujo de costo mínimo con $|f^*| = |f|$. Entonces $f^* - f$ tiene valor cero y costo negativo. Iterativamente encontramos un ciclo en la red con aristas de flujo positivo, obtenemos un flujo f' que es cero siempre salvo en ese camino donde vale lo que la menor arista, y lo restamos. Partimos así $f^* - f$ en una suma de flujos en ciclos. La suma de sus costos es negativa, así que uno de ellos tiene costo negativo. ∎

Teorema *Si f es un flujo de costo mínimo, entonces el flujo que viene de aumentarlo por un camino de aumento de costo mínimo también es de costo mínimo.*

Prueba Sea p el camino y f' el flujo aumentado. Si f' no es mínimo en su red residual hay un ciclo c de costo negativo. Sea $p \oplus c$ el conjunto de aristas que están en p o c menos las que están en p y su reverso está en c (y sus reversos). Se ve que $p \oplus c$ es un camino p' de s a t y algunos ciclos, todos con costo no negativo. Entonces $cost(p') \leqq cost(p \oplus c) = cost(p) + cost(c) < cost(p)$. Contradicción. ∎

Un método para encontrar un flujo máximo de costo mínimo es encontrar un flujo máximo e ir mandando flujo por los ciclos de costo negativo hasta que no haya más. Si las capacidades son enteras hay una cantidad finita de flujos; el método va decrementando costo así que en algún momento llega al mínimo y termina. Esto muestra que si las capacidades son enteras hay un flujo de costo mínimo que además es entero.

Si el grafo no tiene ciclos de costo negativo, el flujo nulo es de costo mínimo así que un método alternativo es ir aumentándolo por caminos de costo mínimo. Si las capacidades son enteras necesitamos como mucho $|f^*|$ iteraciones, cada una de las cuales se reduce a buscar un camino mínimo de s a t. Podría haber costos negativos, así que Dijkstra solo no funciona. En la primera iteración usamos Bellman-Ford y ponemos $cost'(u,v) = cost(u,v) + d(u) - d(v)$. Ahora los costos son no negativos y los del camino de aumento (que es mínimo) son cero, así que en la red residual también, porque solo incorpora aristas del camino de aumento invertidas. La transformación mantiene caminos mínimos, ya que si p es un camino de s a t, $cost'(p) = cost(p) + d(s) - d(t)$. Cuando hacemos Dijkstra en la nueva red hacemos la misma transformación, y así podemos

seguir. Entonces en total pagamos un Bellman-Ford y a lo sumo $|f^*|$ veces Dijkstra, lo que suma $O(nm + (m + n \log n)|f^*|)$.

Teorema *En una red con capacidades enteras y sin ciclos negativos encontramos un flujo máximo de costo mínimo en tiempo* $O(nm + (m + n \log n)|f^*|)$.

1.3.4 Matching máximo

Sea M un matching. Un M-camino de aumento es un camino simple $u_1u_2 \dots u_{k-1}u_k$ tal que u_1 y u_k son libres (no matcheados) y u_2u_3, u_4u_5, $\dots$, $u_{k-2}u_{k-1} \in M$. Sea M^* un matching con $|M^*| > |M|$. Entonces $M' = (M \setminus M^*) \cup (M^* \setminus M)$ está formado por caminos y ciclos pares. Como $|M^* \setminus M| > |M \setminus M^*|$, debe haber un camino que empiece y termine con aristas de M^*; este es un M-camino de aumento. Entonces un matching es máximo si y solo si no tiene caminos de aumento.

Método (Edmonds' blossom shrinking) Para encontrar caminos de aumento largamos un DFS desde los vértices libres mirando intercaladamente aristas libres y matcheadas. Si de un vértice libre se llega a otro hay camino de aumento. Si no, obtenemos un bosque M-alternante, formado por los árboles de búsqueda con raíz en los vértices libres. Haciendo BFS en los árboles distinguimos por niveles vértices pares e impares intercaladamente. Si dos vértices pares de dos árboles distintos son vecinos tenemos un camino de aumento. Si no, supongamos que hay dos vértices pares x, y del mismo árbol de raíz u que son vecinos. El camino xy del árbol y la arista xy forman un ciclo B. Cambiamos aristas matcheadas por no matcheadas en el camino de u a C en el árbol. Ahora x es libre. Contraemos B a un vértice. Veamos que hay un camino de aumento en el grafo contraído si y sólo si hay en el original. Si hay P en el original suponemos que toca a B, sea P' el camino desde un extremo hasta el primer vértice de B que toca; lo extendemos para que llegue a x y tenemos un camino de aumento. Si hay P en el contraído, B es extremo, así que en el original hacemos un camino de x al vértice de B donde sale la primera arista del camino y obtenemos un camino de aumento. Entonces buscar un camino de aumento en el original es buscar uno en el contraído. Supongamos que no quedan blossoms por reducir y no encontramos dos vértices pares vecinos. Entonces el conjunto de vértices pares forma un conjunto independiente y tanto los impares como los demás están matcheados, así que el matching es máximo. Esto hace un algoritmo de tiempo $O(n^2m)$: son $O(n)$ búsquedas de caminos de aumento; en cada una se contraen $O(n)$ blossoms, y cada contracción o descontracción es de tiempo lineal $O(m)$.

Algoritmo (Gabow) Mantenemos una disjoint set structure para guardar los blossoms. La estructura soporta makeset, link y find. Al principio hacemos makeset con cada vértice. Mantenemos $origin : V \to V$, que dice la base del blossom en el que está cada vértice o sí mismo. Si $uv \in E$, $u'v' \in E'$, donde $u' = origin(\text{find}\, u)$. Tenemos $mate : V \to V \cup \{\text{null}\}$ que mapea vértices con sus matcheados o con null si son libres. Mantenemos $label : V \to \{\text{unreached}, \text{even}, \text{odd}\}$, y $pred : V \to V$, que dice el padre de los vértices impares. Entonces $p : V \to V$, el padre de u en el bosque, se calcula como, si u es unreached, u; si es par, $mate(u)$; si es impar, $pred(u)'$. Si un vértice impar u se contrae, ponemos $bridge(u) = (v, w)$, donde v es descendiente de u par y w es el otro par del árbol al que se conectó v. (Esto sirve para encontrar un camino a la raíz del árbol desde u tal que de u sale una arista matcheada: se va de u a v, de v a w, y se sube por w.) Esto nos permite, si encontramos un vértice vw con v' y w' pares, determinar si son parte del mismo árbol, contraer el blossom, o encontrar el camino de aumento, si no. En efecto, miramos la secuencia $v_0, w_0, v_1, w_1, \dots$, dada por $v_0 = v', w_0 = w'$, $v_{i+1} = p(v_i), w_{i+1} = p(w_i)$; si llegamos a dos vértices libres diferentes, x, y, guardamos v, w, x, y y paramos; si nos encontramos a uno común u, encontramos un blossom $v_0, v_1, \dots, v_j = u$, $w_0, w_1, \dots, w_k = u$; para $i = 0, \dots, j-1$ hacemos link (find (u), find (v_i)) y si v_i es impar

ponemos $bridge(v_i) = (v, w)$; para $i = 0, \ldots, k-1$ hacemos link (find (u), find (w_i)) y si w_i es impar ponemos $bridge(w_i) = (w, v)$; ponemos $origin(\text{find}(u)) = u$. El camino de aumento en base a v, w, x, y es $\text{reverse}(path(v, x)) \,\&\, path(w, y)$, donde $path(v, w)$ se define recursivamente así: si $v = w$ es $[v]$, y si v es par, es $[v, mate(v)] \,\&\, path(pred(mate(v)), w)$; si v es impar, es $[v] \,\&\, \text{reverse}(path(x, mate(v))) \,\&\, path(y, w)$, donde (x, y) es $bridge(v)$. El algoritmo consiste en primero inicializar todos los vértices poniendo $origin(v) = v$, makeset(v) y poniendo $label(v) =$ even si es libre y $label(v) =$ unreached si está matcheado. Ahora vamos tomando una por una todas las aristas vw con v' par (lo que hacemos es al principio ir metiéndolas en una lista; cuando contraemos un blossom agregamos las vecinas de los vértices impares); si w' es impar no hacemos nada, si w' es unreached lo hacemos impar y $mate(w')$ par, con $pred(w') = v$; si w' es par ya vimos qué hacer.

Visitar las aristas cuesta $O(m)$, contraer cada blossom es en total $O(n)$ igual que encontrar las bases del camino de aumento, sin contar n makeset, como mucho $n-1$ link y $O(m)$ find, que cuestan en total $O(m\alpha(m, n))$. Como se encuentran a lo sumo $O(n)$ caminos de aumento, la complejidad total es de $O(nm\alpha(m, n))$.

Teorema (Tutte-Berge) *El cardinal de un matching máximo en un grafo G es*[19]

$$\nu(G) = \min_{X \subset V}(|V| + |X| - o(G \setminus X))/2.$$

Prueba Una implicación es fácil. Si no hay matching perfecto largamos el algoritmo de Edmonds. Hacemos que X sea el conjunto de vértices impares que no están en un blossom. Ahora $G \setminus X$ tiene exactamente $|X| + n - 2|M|$ blossoms o vértices pares aislados, y están todos desconectados entre sí; el resto tiene un matching perfecto así que las componentes son pares. ∎

Podemos obtener más información todavía. Sea A el X antes, D los pares y el resto de los impares, y C el resto. Vemos que D es el conjunto de los vértices v tales $\nu(G) = \nu(G \setminus v)$, $A = N(D)$, $C = V \setminus D \setminus A$, D es la unión de las componentes impares de $G \setminus A$, toda componente de D es factor critical (o sea que $G \setminus v$ tiene un matching perfecto para todo v), $B = A \cup C$ es el conjunto de vértices cubiertos por todo matching máximo, A es el subconjunto de B de vértices con al menos un vecino fuera de B; todo matching máximo es un matching perfecto de C, un matching de A en las distintas componentes de D y matchings casi perfectos para las componentes de D no matcheadas. La descomposición es única y se llama *descomposición de Edmonds-Gallai*.

1.3.5 Ejercicios

1. (Qualification Round) Consideramos un conjunto A de N elementos y una familia de subconjuntos $A_1, A_2, ..., A_P$. Dados los cardinales $s_i = |A_i|$ y un número C queremos elegir N y los subconjuntos de manera que la cantidad de elementos de A que están en al menos C de los A_i sea máxima. Encontrar esa cantidad máxima en tiempo $O(P \log P \log(\sum_{i=1}^{P} s_i/C))$.

2. Dado un grafo G, funciones $a, b : V \to \mathbb{R}^+$ y $w : E \to \mathbb{R}^+$, se quiere una partición (A, B) de V tal que

$$\sum_{v \in A} a(v) + \sum_{v \in B} b(v) + \sum_{\substack{uv \in E \\ u \in A, v \in B}} w(uv)$$

sea mínimo. Reducir el problema a encontrar un mínimo corte.

[19] Si G es un grafo, $o(G)$ es su cantidad de componentes conexas de tamaño impar.

3. Sea $w : [N] \to \mathbb{R}^+$, $\mathcal{F}$ un conjunto de N subconjuntos de $[N]$, y $b : \mathcal{F} \to \mathbb{R}^+$ una función. Encontrar el subconjunto $\mathcal{X}$ de $\mathcal{F}$ que maximice

$$\sum_{X \in \mathcal{X}} b(x) - \sum_{i \in \bigcup \mathcal{X}} w(i)$$

en tiempo $O((N+M)^3)$.

4. Se tienen N jugadores que compiten todos contra todos, una vez cada par; quien gana en cada partida se anota un punto. Es dada una secuencia de reales positivos $a_1, \ldots, a_N$ y se pide devolver otra, $b_1, \ldots, b_N$, que resulte de una posible competición como la recién descripta, de manera que se minimice el número $\sum_{i=1}^{N} |a_i - b_i|$. El tiempo no sé si es $O(N^3)$ o $O(N^4)$.

5. Sea V un conjunto y $i, o : V \to \mathbb{N}_0$ funciones. Decidir si existe un grafo dirigido (V, E) tal que para todo $u \in V$ el indegree de u sea $i(u)$ y el outdegree, $o(u)$, reduciendo el problema a buscar un flujo máximo.

6. Reducir el problema de encontrar el mínimo corte con la menor cantidad de aristas en una red a encontrar un mínimo corte.[20]

7. Se tienen m experimentos $E_1, \ldots, E_m$ y n instrumentos $I_1, \ldots, I_n$; cada experimento requiere algunos de esos instrumentos. Hacer el experimento E_i paga p_i, pero tener que comprar el instrumento I_j cuesta c_j. Elegir experimentos de manera que se maximice la suma de las pagas menos la suma de los costos.[21]

8. Hay n estudiantes, m proyectos, t supervisores. Cada proyecto tiene un máximo de estudiantes b_i que pueden elegirlo. Cada proyecto tiene un conjunto A_i de profesores que pueden supervisarlo. Cada profesor puede supervisar a lo más k_j chicos. A cada chico se le asigna exactamente un proyecto. Los chicos rankearon los proyectos que les gustan en orden $1, \ldots, m$, o sea r_{ij} es el orden del proyecto j en el ranking del estudiante i. Asignar $p : [n] \to [m]$ a cada estudiante un proyecto satisfaciendo las demandas. Además la suma $\sum_{i=1}^{n} r_{ip(i)}$ tiene que ser mínima.

9. (Menger) Probar que si u, v son vértices de un grafo (dirigido o no) entonces la mayor cantidad de caminos $u \rightsquigarrow v$ que no comparten aristas es igual a la menor cantidad de aristas necesaria para desconectar u y v.

10. (Menger) Probar que si u, v son vértices no vecinos en un grafo (dirigido o no) entonces la mayor cantidad de caminos $u \rightsquigarrow v$ que no comparten vértices es igual a la menor cantidad de vértices necesaria para desconectar u y v.

11. (Escape problem) Dado un grafo G, sean $A = \{a_1, \ldots, a_k\}, B$ dos subconjuntos de V. Decidir si existen caminos disjuntos $a_1 \rightsquigarrow b_1, \ldots, a_k \rightsquigarrow b_k$, con $b_1, \ldots, b_k \in B$ en tiempo $O(mk)$.

12. (Minimum path cover in a DAG) Dado un grafo dirigido sin ciclos, un cubrimiento por caminos disjuntos es un conjunto de caminos disjuntos en vértices (posiblemente compuestos por un solo vértice) tales que todo vértice está en un camino. Encontrar el mínimo cubrimiento usando flujo máximo.[22]

13. (2-factor) Dado un grafo (dirigido o no) determinar si sus vértices se pueden partir en ciclos disjuntos en tiempo $O(mn)$.

---

[20] Sugerencia. Cambiar las capacidades por $c' : (u, v) \mapsto (E+1)c(u, v) + 1$.

[21] Sugerencia. Mirar la red $V = \{E_1, \ldots, E_m\} \cup \{I_1, \ldots, I_n\} \cup \{s, t\}$, con $c(s, I_k) = c_k$, $c(E_j, t) = p_j$ y si E_j usa I_k entonces $c(I_k, E_j) = \infty$.

[22] Sugerencia. Si $G = (V, E)$ es el grafo, con $V = \{1, \ldots, n\}$, considerar el grafo $G' = (V', E')$ dado por $V' = \{x_0, x_1, \ldots, x_n, y_0, y_1, \ldots, y_n\}$, $E' = \{(x_0, x_i) \mid 1 \leqq i \leqq n\} \cup \{(y_i, y_0) \mid 1 \leqq i \leqq n\} \cup \{(x_i, y_j) \mid (i, j) \in E\}$.

14. (Gallai-Milgram) En un grafo dirigido sin ciclos, el mínimo cubrimiento por caminos disjuntos tiene como mucho tantos elementos como el máximo conjunto independiente[23].

15. (Dilworth) En un grafo dirigido sin ciclos tal que $uv, vw \in E$ implican $uw \in E$, el mínimo cubrimiento por caminos disjuntos tiene exactamente la misma cantidad de elementos que un máximo conjunto independiente.

16. Un grafo dirigido sin ciclos con al menos $rs + 1$ elementos contiene un camino de largo $r + 1$ o un conjunto independiente de cardinal $s + 1$.

17. (Erdős-Szekeres) En una secuencia de al menos $n^2 + 1$ números hay una subsecuencia monótona de tamaño $n + 1$.

18. (Min-max feasible flow) Sea N una red de flujo y sea $l : E \to \mathbb{R}_0^+$. Un flujo es feasible si satisface que $l(u, v) \leqq f(u, v)$ para todo $(u, v) \in E$. Reducir el problema de encontrar el máximo y mínimo flujo feasible a dos máximo flujo.[24]

19. (Original) Sea N una red de flujo con varios sources, sea s uno de ellos. Se quiere encontrar un flujo máximo en N que maximice el flujo que sale de s.[25]

20. Probar que todo grafo bipartito regular tiene un matching perfecto.

21. (Sperner) Demostrar que la máxima cantidad de subconjuntos de $[n]$ tales que ninguno está contenido en otro es $\binom{n}{\lfloor n/2 \rfloor}$.[26]

22. (Kőnig) La mínima cantidad de colores necesaria para pintar las aristas de un grafo bipartito sin que dos del mismo color sean adyacentes es el máximo grado de un vértice.

23. (Birkhoff-Von Neumann) Una matriz en $\mathbb{R}_{\geqq 0}^{n \times n}$ se dice doblemente estocástica si sus filas y columnas suman uno. Demostrar que toda matriz doblemente estocástica A se puede escribir como combinación convexa de matrices de permutaciones, es decir, que hay $\lambda : S_n \to [0, 1]$ con $A = \sum_{\sigma \in S_n} \lambda_\sigma P_\sigma$, con $\sum_{\sigma \in S_n} \lambda_\sigma = 1$.[27]

24. (f-factor en bipartitos) Dado un grafo bipartito G y una función $f : V \to \mathbb{N}_0$, decidir si G tiene un subgrafo G' con $V(G') = V(G)$ y $\deg_{G'}(u) = f(u)$ para todo $u \in V$ usando máximo flujo.

25. (f-matching en bipartitos) Dado un grafo bipartito G y una función $f : V \to \mathbb{N}_0$, decidir si hay $w : E \to \mathbb{N}_0$ con $\sum_{uv \in E} w(uv) = f(u)$ para todo $u \in V$ usando máximo flujo.

26. (f, g-factor en digrafos) Dado un digrafo G y funciones $f, g : V \to \mathbb{N}_0$, decidir si G tiene un subgrafo G' con $V(G') = V(G)$, $\deg^+_{G'}(u) = f(u)$ y $\deg^-_{G'}(u) = g(u)$ para todo $u \in V$ usando máximo flujo.

27. (f-orientación) Dado un grafo G y una función $f : V \to \mathbb{N}_0$, decidir si existe una orientación D de las aristas tal que $\deg^+_D(u) = f(u)$ para todo $u \in V$ usando máximo flujo.

---

[23] También vale si el grafo tiene ciclos. Es fácil, pero no sale con flujo.

[24] Sugerencia. Buscar un flujo feasible f se reduce a buscar un flujo máximo en la red dada con dos vértices más x e y (source y sink), con capacidades c' dadas por $c' = c - l$ donde tiene sentido, $c'(t, s) = \infty$, $c'(x, u) = \sum_{vu \in E} l(v, u)$ y $c'(u, y) = \sum_{uv \in E} l(u, v)$. Para buscar el flujo mínimo o máximo miramos la red residual y corregimos f.

[25] Sugerencia. (Agustín) Agregar vértices x, y, separar s en s^-, s^+, conectar x a s^+ y s^- a y con capacidad infinita.

[26] Sugerencia. Si $0 \leqq k < n/2$, en el grafo bipartito con vértices $\{A \subset [n] \mid |A| = k\}$ y $\{B \subset [n] \mid |B| = k + 1\}$ y $AB \in E$ si y solo si $A \subset B$ hay un matching perfecto.

[27] Sugerencia. Una posibilidad es probar primero que hay $\sigma \in S_n$ con $A_{i\sigma(i)} > 0$ para $i = 1, \ldots, n$. Otra es probar primero el resultado para racionales, multiplicando por los denominadores para que queden todos números enteros, y luego extender al caso real con un argumento de compacidad.

28. (Folkman-Fulkerson) Sea G con $V = (A, B)$ un grafo bipartito. Entonces G tiene h matchings disjuntos en aristas de tamaño t si y sólo si para todos $X \subset A$ y $Y \subset B$, $\{uv \in E \mid u \in X, v \in Y\} \geqq h(t - |A - X| - |B - Y|)$.[28]

29. (Tarjan) En una red tal que las capacidades de sus aristas son uno el algoritmo de Dinic tarda tiempo $O(\text{mín}\{n^{2/3}, m^{1/2}\}m)$.

30. (Assignment Problem) Dado un grafo bipartito pesado encontrar un matching de peso máximo en tiempo $O(nm + n^2 \log n)$.

31. (General min cost flow) Dado un grafo dirigido conexo G y funciones $l, u, c : E \to \mathbb{R}$ con $0 \leqq l \leqq u$ y $b : V \to \mathbb{R}$ con $\sum_{v \in V} d(v) = 0$ se busca una función $f : E \to \mathbb{R}$ tal que $l \leqq f \leqq u$, $\sum_{vu \in E} f(vu) - \sum_{uv \in E} f(uv) = b(u)$ y $\sum_{e \in E} c(e)f(e)$ sea mínimo. Reducir este problema a buscar un flujo máximo de costo mínimo en una red.

32. (Napkin problem) En N días consecutivos se necesita usar $r_1, \ldots, r_N$ servilletas. Comprar una cuesta a. Se pueden lavar algunas servilletas usadas a un costo de b cada una y usarlas al día siguiente, o a un costo c y usarlas a los dos días. Determinar el mínimo costo que se debe pagar para tener la cantidad de servilletas deseada en tiempo $O((r_1 + \cdots + r_n)n \log n)$.

33. (f-matching) Dado un grafo y una función $f : V \to \mathbb{N}_0$ encontrar $w : E \to \mathbb{N}_0$ con $\sum_{uv \in E} w(uv) \leqq f(u)$ para todo $u \in V$ y $\sum_{e \in E} w(e)$ máxima en tiempo $O(nmF^3\alpha(nF, mF))$, donde $F = \max_{u \in V} f(u)$.[29]

34. (f-factor) Dado un grafo G y una función $f : V \to \mathbb{N}_0$ encontrar $G' \subset G$ con $V(G') = V(G)$ tal que $\deg_{G'} \leqq f$ con cantidad de aristas máxima en tiempo $O(m^3\alpha(n, m))$.[30]

35. (Stable Marriage) Se tiene un conjunto chicos A y uno de chicas B. A cada chico le gustan algunas chicas, en un orden de preferencia; lo mismo las chicas con los chicos. Encontrar un matching $M \in A \times B$ tal que si $ab \notin M$ entonces a está matcheado con una chica que le gusta más que b o b está matcheada con un chico que le gusta más que a en tiempo $O(|A||B|)$.

[28] Sugerencia. Agregar $|A| - t$ vértices a B y $|B| - t$ vértices a A, con infinitas aristas que salen de cada uno de los nuevos vértices a todos los del otro color. Entonces la condición es que este grafo tenga un h-factor.

[29] Sugerencia. Multiplicar cada vértice u $f(u)$ veces; unir $u'v' \in E'$ si y sólo si $uv \in E$, donde u' es clon de u.

[30] Sugerencia. Si partimos las aristas con dos vértices, o sea sacamos $e = uv$ y ponemos ux, xy, yv, y hacemos $f = 1$ en los nuevos vértices, el problema se reduce a buscar un f-matching.