Diseño y Análisis de Algoritmos
Apuntes

Gonzalo Navarro

Figuras y Apéndices por Manuel Ariel Cáceres

Departamento de Ciencias de la Computación
Universidad de Chile

Índice general

Capítulo 1

Cotas Inferiores

La *complejidad de un problema* es el costo del *mejor algoritmo* que resuelve ese problema. Por lo tanto, cada nuevo algoritmo que se encuentra para resolver ese problema establece una nueva *cota superior* a su complejidad. Por ejemplo, si descubrimos el ordenamiento por inserción, que toma tiempo $O(n^2)$, podemos decir que la complejidad del problema de ordenar es $O(n^2)$. Luego descubrimos MergeSort, y eso nos permite decir que la complejidad del problema de ordenar es en realidad menor, $O(n \log n)$.

¿Cómo podemos determinar que hemos encontrado el algoritmo *óptimo* (el de menor costo posible) para resolver un problema? Debemos ser capaces de conocer su complejidad, pero mediante encontrar algoritmos sólo podremos establecer cotas superiores. Necesitamos entonces mecanismos para establecer *cotas inferiores* a un problema, es decir, demostrar que *cualquier* algoritmo que resuelva ese problema debe al menos pagar un determinado costo.

Por ejemplo, podemos convencernos de que para ordenar, el algoritmo debe al menos examinar todos los elementos del arreglo, para lo cual necesita realizar n accesos. Eso significa que ningún algoritmo de ordenamiento puede tener costo $o(n)$. Eso lo expresamos diciendo que una cota inferior para el problema de ordenar es $\Omega(n)$. Note que esta cota inferior es válida, pero podría no ser (de hecho, no es), la mejor posible (la más alta posible). Decimos que esta cota puede no ser *ajustada*.

Para algunos problemas, se conocen cotas superiores de $O(T(n))$ y a la vez cotas inferiores de $\Omega(T(n))$. En ese caso, sabemos que

- Los algoritmos que toman tiempo $O(T(n))$ son óptimos, es decir, no pueden existir algoritmos de complejidad inferior que resuelvan el problema.
- La cota inferior de $\Omega(T(n))$ es ajustada, es decir, no puede haber una mejor cota inferior para el problema.
- Conocemos la complejidad exacta del problema, que expresamos diciendo que el problema tiene complejidad $\Theta(T(n))$.

Para los problemas en que esto no ocurre, tenemos una cota superior (es decir, la complejidad de un algoritmo que lo resuelve) de $O(T_1(n))$ y una cota inferior de $\Omega(T_2(n))$, con $T_2(n) = o(T_1(n))$. No sabemos si el algoritmo es óptimo, ni si la cota inferior es ajustada, y no conocemos la complejidad exacta del problema.

Se puede hablar de estas complejidades entendiendo que nos referimos al *peor caso* de un algoritmo que resuelva el problema, que es lo más común, pero también interesa conocer la complejidad de caso promedio de un problema, dada una cierta distribución de los inputs posibles. En el caso de algoritmos aleatorizados, podemos hablar de la complejidad esperada de un problema, que considera el peor input posible pero promedia sobre las elecciones aleatorias que hace el algoritmo. En este capítulo, sin embargo, sólo consideraremos algoritmos determinísticos (no aleatorizados).

Asimismo, nos puede interesar la complejidad no en términos asintóticos, sino en términos del costo exacto de los algoritmos (cantidad exacta de comparaciones que hace, o de accesos a un arreglo, etc.).

Note que una cota inferior puede centrarse en contar sólo cierto tipo de operaciones e ignorar otras, y aún así será válida como cota inferior.

En este capítulo veremos tres técnicas básicas para establecer cotas inferiores: la estrategia del adversario, reducciones, y teoría de la información.

1.1. Estrategia del Adversario

La *estrategia del adversario* se usa para demostrar cotas inferiores de peor caso. La idea es demostrar que, para responder correctamente frente a cualquier input, el algoritmo necesita aprender lo suficiente sobre el input, y para ello debe pagar un cierto costo. La figura del adversario se utiliza como metáfora del peor caso. El adversario actúa como intermediario entre el algoritmo y el input. Cada vez que el algoritmo paga el costo de preguntar algo sobre el input, el adversario decide qué responder. Es decir, el input no existe de antemano, sino que el adversario lo va construyendo de modo de provocar el costo más alto posible al algoritmo. Su única restricción es que lo que responde debe ser consistente, es decir, debe existir algún input cuyas respuestas a las preguntas del algoritmo sean las mismas que las del adversario.

Un ejemplo ilustrativo es el juego de las 20 preguntas. Éste consiste en que un jugador A piensa en un personaje X y el otro jugador B debe adivinar el personaje mediante hacer a lo sumo 20 preguntas a A, de respuesta sí/no. Aquí B es el algoritmo y A actúa como adversario, haciendo de interfaz entre B y el personaje que tiene en mente. ¿Alguna vez jugó a este juego? ¿Se le ocurrió, siendo A, no decidir X de entrada sino irlo definiendo a medida que B preguntaba, para asegurarse de que B nunca llegara a una respuesta correcta en 20 preguntas? ¿De qué debe cuidarse si hace esto?

En general, para aplicar este método requerimos crear un *modelo* de lo que el algoritmo va aprendiendo acerca del input. El modelo debe tener un *estado inicial*, que corresponde al

comienzo de la ejecución, cuando el algoritmo no sabe nada del input. Debe tener uno o más *estados finales*, cuando el algoritmo aprendió lo suficiente del input como para responder correctamente. Y el mínimo costo de llegar del estado inicial a algún estado final es una cota inferior al costo del algoritmo: no importa qué algoritmo sea, este modelo es válido e implica que el algoritmo no puede responder siempre correctamente si no paga un cierto costo. El rol del adversario es elegir el peor resultado posible (que el algoritmo avance lo menos posible hacia un estado final) por cada acción que realiza. Veremos la forma de aplicar esta técnica a lo largo de varios ejemplos.

1.1.1. Búsqueda en un arreglo

Como un caso muy simple, que ni siquiera requiere de un modelo, consideremos el problema de encontrar dónde está un determinado elemento en un arreglo desordenado (supongamos que sabemos que está, pero no dónde). Para resolver este problema, cualquier algoritmo debe examinar las n celdas del arreglo, pues si un algoritmo dejara alguna celda sin leer, el adversario colocaría allí al elemento buscado. Es decir, el adversario responde con elementos distintos al buscado cada vez que el algoritmo accede a una celda del arreglo, a menos que sea la última celda restante. Esto es una abstracción del hecho de que, sea cual sea la estrategia del algoritmo para examinar las celdas, hay un input en el cual el elemento estará en una de las celdas no examinadas, por lo cual en el peor caso es necesario examinarlas todas.

Arreglo ordenado. Si el arreglo está ordenado, el adversario ya no puede obligar al algoritmo a examinar todas las celdas, pues debe ser consistente: si entrega un determinado elemento $A[i] = x$, entonces debe entregar elementos $\leq x$ en celdas $< i$, y elementos $\geq x$ en celdas $> i$. Por ello, no tenemos una cota inferior de n comparaciones en este caso.

Para encontrar una cota en el caso ordenado, usaremos el siguiente modelo. Todo algoritmo realiza una serie de accesos a A hasta responder, y contaremos sólo la cantidad de esos accesos como su costo. El modelo es que el algoritmo sabe que el elemento buscado tiene que estar en un rango $A[i, j]$ del arreglo original $A[1, n]$. Mediante acceder a un elemento $A[k]$, puede ser que: (1) $k \notin [i, j]$, en cuyo caso el algoritmo no aprende nada; (2) $A[k]$ es el elemento buscado, en cuyo caso el algoritmo ahora sabe que el elemento está en el rango $A[k, k]$; (3) $A[k]$ sea mayor que el elemento buscado, en cuyo caso el algoritmo ahora sabe que el elemento está en el rango $A[i, k-1]$; y (4) $A[k]$ sea menor que el elemento buscado, en cuyo caso el algoritmo ahora sabe que el elemento buscado está en el rango $A[k+1, j]$.

El estado inicial es $A[1, n]$ y los estados finales son todos los rangos $A[k, k]$, $1 \leq k \leq n$. Podemos ver por inducción que el algoritmo nunca ha mirado un elemento dentro del rango actual $A[i, j]$, por lo cual el adversario es libre de decidir entre las alternativas (2), (3) y (4) (el algoritmo elige el k, por lo cual nunca cometerá la tontería de elegir (1)). El adversario intentará que el rango se mantenga lo mayor posible, pues al llegar a tamaño 1 el algoritmo llega a un estado final. Por ello, nunca elegirá (2). Elegirá (3) si $k - i \geq j - k$ y (4) si no. Esto garantiza que el intervalo se reduce a lo sumo a la mitad en cada iteración (cuando tiene

largo par), por lo cual cualquier algoritmo requiere en el peor caso $\lfloor \log_2 n \rfloor$ comparaciones (requiere una más si no se sabe si el elemento está en A).

Cotas superiores. En ambos casos, arreglo desordenado y ordenado, sabemos que las cotas inferiores son ajustadas porque conocemos la búsqueda secuencial y binaria, que dan cotas superiores iguales a las inferiores. Pero en general este método no tiene por qué dar cotas inferiores ajustadas. Por ejemplo, puede que hayamos elegido un modelo que no representa todo lo que el algoritmo debe aprender sobre el input para contestar correctamente, por lo que le permite llegar del estado inicial a uno final a un costo inferior al del algoritmo óptimo. También puede que el adversario no sea lo suficientemente inteligente y no fabrique el peor input posible para el algoritmo.

En el caso del arreglo ordenado se nota otro aspecto importante de esta técnica: suele sugerir lo que debería hacer un algoritmo óptimo. En este caso, nos queda claro que lo mejor es consultar a la mitad del intervalo, pues de otro modo el adversario hará que nuestro intervalo se reduzca más lentamente. Es decir, nos sugiere el algoritmo de búsqueda binaria.

1.1.2. Máximo de un arreglo

Consideremos el problema de buscar el máximo elemento en un arreglo $A[1, n]$ mediante comparaciones. Es decir, los elementos de A son cajas negras donde la única operación que podemos hacer sobre ellos es compararlos. Para simplificar, supondremos que todos los elementos son distintos.

El algoritmo más simple es tomar $A[1]$ como máximo provisional, y luego comparar ese máximo con $A[2]$, $A[3]$, y así hasta $A[n]$, manteniendo siempre el máximo visto hasta ahora. Para un arreglo $A[1, n]$ requerimos entonces $n - 1$ comparaciones.

Un algoritmo alternativo es el llamado "torneo de tenis": Comparamos $A[1]$ con $A[2]$, comparamos $A[3]$ con $A[4]$, $A[5]$ con $A[6]$, etc. Luego, en un nuevo arreglo de los $n/2$ ganadores (mayores que el otro) en las comparaciones, volvemos a hacer una ronda de comparar a cada impar con el par que le sigue, y continuamos hasta tener un único ganador. Las comparaciones forman un árbol binario donde las n hojas son los elementos de A y cada nodo interno es una comparación, por lo que se hacen también $n - 1$ comparaciones.

Cota inferior. Veamos que efectivamente se necesitan $n-1$ comparaciones para encontrar el máximo. Usaremos el modelo siguiente. El conocimiento que algoritmo tiene sobre el input es un grafo de n nodos $(1), \ldots, (n)$. Cada vez que el algoritmo compara dos elementos $A[i]$ y $A[j]$, agregamos una arista entre los nodos (i) y (j). El estado inicial es el grafo sin aristas.

Para establecer los estados finales, notemos que un grafo no conexo no puede ser final. Si no existe un camino entre (i) y (j), el algoritmo no puede saber cuál es mayor a partir de las comparaciones realizadas, incluso aplicando transitividad. El adversario puede decidir que todos los elementos de la componente conexa de (i) son mayores o menores que todos los de la de (j) sin violar ninguna de las respuestas que ya ha dado. Por lo tanto, si el algoritmo

declara que el máximo es un determinado $A[i]$, existe un input para el cual el resultado es incorrecto si existe un $A[j]$ en otra componente conexa.

Por lo tanto, los estados finales deben ser grafos conexos. Como se necesitan al menos $n-1$ aristas para conectar un grafo de n nodos, $n-1$ es una cota inferior al problema de encontrar el máximo de un arreglo. Como tenemos algoritmos que usan $n-1$ comparaciones, sabemos que son óptimos y que esta cota inferior es ajustada, a pesar de que sólo consideramos una condición bastante débil acerca de lo que debe conocer el algoritmo sobre el input: nos bastó que existiera un camino de comparaciones que conectara a cualquier par de elementos.

Otro modelo. Consideremos ahora un modelo completamente distinto. El conocimiento del algoritmo sobre el input se describirá con tres variables (a, b, c):

- a es el cardinal del conjunto A de los elementos que nunca han sido comparados (no confundir con el arreglo A);
- b es el cardinal del conjunto B de los elementos que se han comparado alguna vez y han ganado (han resultado mayores) en todas sus comparaciones; y
- c es el cardinal del conjunto C de los elementos que han perdido (han resultado menores) en alguna comparación.

El estado inicial es $(n, 0, 0)$, y está claro que el estado final debe ser $(0, 1, n-1)$, pues si $a > 0$ hay un elemento sin comparar (y el adversario se encargará de que ése sea el máximo) y si $b > 1$ hay dos elementos que han ganado todas sus comparaciones (y si el algoritmo declara ganador a uno de ellos el adversario puede decidir que es menor que el otro). Cada comparación mueve elementos dentro de la tupla (a, b, c). Según de qué conjunto vengan los elementos que el algoritmo compara, la siguiente tabla indica los posibles nuevos estados a partir de un estado (a, b, c):

	A	B	C
A	$(a-2, b+1, c+1)$	$(a-1, b, c+1)$	$(a-1, b+1, c)$ $(a-1, b, c+1)$
B		$(a, b-1, c+1)$	(a, b, c) $(a, b-1, c+1)$
C			(a, b, c)

Por ejemplo, el resultado de la celda $\langle A, A\rangle$ es el de comparar dos elementos que nunca habían sido comparados: ambos salen del conjunto A, uno pasa a haber ganado todas sus comparaciones (B) y otro a haber perdido alguna comparación (C), por lo tanto el nuevo estado es $(a-2, b+1, c+1)$. El caso $\langle A, B\rangle$ nos lleva a $(a-1, b, c+1)$ independientemente de que el ganador sea el elemento de A o el de B (si el de A gana, pasa a estar en B pero el que estaba en B pasa a C). El caso $\langle A, C\rangle$ puede llevar a dos estados distintos dependiendo de quién gane la comparación. Como el elemento de A nunca se había comparado, el adversario puede

decidir cuál de los dos resultados es el que ocurrirá. En particular, el adversario podría elegir la estrategia de "los que han perdido siguen perdiendo", con lo cual elige $(a-1, b+1, c)$ en este caso, y elige (a, b, c) para el caso $\langle B, C\rangle$ (pues el adversario puede hacer que un elemento de B sea tan grande como desee).

En cualquier caso, podemos observar que c crece a lo sumo de a uno, y como debe pasar de 0 a $n-1$, se necesitan al menos $n-1$ comparaciones para llegar al estado final.

Esta técnica nos sugiere algoritmos óptimos más claramente que la del grafo. Primero es necesario comparar $\langle A, A\rangle$. Luego, manteniendo $b = 1$, podemos seguir siempre comparando $\langle A, B\rangle$ (es decir, el único que ha ganado siempre contra uno que no se ha comparado), para mover los otros $n-2$ elementos de A a C. Puede verse que esta es la estrategia de nuestro primer algoritmo. Alternativamente, podemos utilizar $\langle A, A\rangle$ $n/2$ veces, hasta vaciar A y tener $b = n/2$, y luego comparar $\langle B, B\rangle$ $n/2-1$ veces hasta dejar un solo elemento en B y el resto en C. Note que en la segunda fase se comparan siempre ganadores con ganadores, lo que es compatible con nuestro algoritmo del torneo de tenis. Está claro que ningún algoritmo debería volver a comparar elementos de C, pues el adversario podría hacer que no avance hacia el estado final.

1.1.3. Mínimo y máximo de un arreglo

Consideremos ahora el problema de encontrar el mínimo y el máximo de un arreglo $A[1, n]$. Una forma de resolver este problema es encontrar el máximo de $A[1, n]$ usando $n-1$ comparaciones, y luego, excluyendo el máximo, encontrar el mínimo de los $n-1$ elementos restantes usando $n-2$ comparaciones. En total este algoritmo realiza $2n-3$ comparaciones.

Para ver si es óptimo, usaremos un modelo que extiende el que acabamos de usar, dividiendo el conjunto en cuatro:

- a es el cardinal del conjunto A de los elementos que nunca han sido comparados (nuevamente, no confundir con el arreglo A);
- b es el cardinal del conjunto B de los elementos que se han comparado alguna vez y han ganado (han resultado mayores) en todas sus comparaciones;
- c es el cardinal del conjunto C de los elementos que se han comparado alguna vez y han perdido (han resultado menores) en todas sus comparaciones; y
- d es el cardinal del conjunto D de los elementos que han ganado alguna vez y también han perdido alguna vez.

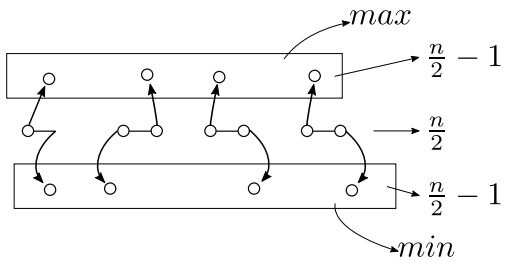
El estado inicial es ahora $(a, b, c, d) = (n, 0, 0, 0)$, y el estado final es $(0, 1, 1, n-2)$. La tabla de resultados de comparaciones es ahora la siguiente:

	A	B	C	D
A	$(a-2, b+1, c+1, d)$	~~$(a-1, b, c, d+1)$~~ $(a-1, b, c+1, d)$	$(a-1, b+1, c, d)$ ~~$(a-1, b, c, d+1)$~~	$(a-1, b+1, c, d)$ $(a-1, b, c+1, d)$
B		$(a, b-1, c, d+1)$	(a, b, c, d) ~~$(a, b-1, c-1, d+2)$~~	(a, b, c, d) $(a, b-1, c, d+1)$
C			$(a, b, c-1, d+1)$	$(a, b, c-1, d+1)$ (a, b, c, d)
D				(a, b, c, d)

Hemos tachado resultados que el adversario podría evitar siempre con la estrategia de "los ganadores siguen ganando y los perdedores siguen perdiendo", que siempre es consistente. Podríamos tachar otros resultados, pero no es necesario para establecer nuestra cota inferior. Observe que (1) a decrece a lo sumo de a dos, (2) d crece a lo sumo de a uno, y (3) a nunca decrece al mismo tiempo que d crece. Entonces, como a debe pasar de n a 0 y d debe pasar de 0 a $n-2$, tenemos una cota inferior de $\lceil \frac{3}{2}n \rceil - 2$ comparaciones para resolver este problema.

Esta vez nuestra cota inferior es distinta de la cota superior $2n-3$. Nos podemos preguntar si será una cota inferior ajustada. Tal vez el modelo es muy débil o el adversario no es muy inteligente o nuestra observación sobre cómo llegar del estado inicial al final no es suficiente para demostrar que realmente se necesitan $2n-3$ comparaciones.

Veremos que no es así, usando el modelo para encontrar un algoritmo óptimo. La tabla sugiere que la forma más rápida de llegar al estado final es usar celdas $\langle A, A\rangle$ $n/2$ veces hasta llegar al estado $(0, n/2, n/2, 0)$. Luego podemos usar $\langle B, B\rangle$ $n/2-1$ veces para llegar a $(0, 1, n/2, n/2-1)$ y finalmente usar $\langle C, C\rangle$ $n/2-1$ veces para llegar al estado final, $(0, 1, 1, n-2)$. Esto equivale a realizar un primer nivel de torneo de tenis, comparando cada celda impar con la siguiente celda par, obteniendo $n/2$ ganadores y $n/2$ perdedores. Luego, buscamos (con cualquiera de los algoritmos vistos) el máximo entre los $n/2$ ganadores y el mínimo entre los $n/2$ perdedores. El costo total es $\lceil \frac{3}{2}n \rceil - 2$ (note que se necesitan dos comparaciones más si n es impar).

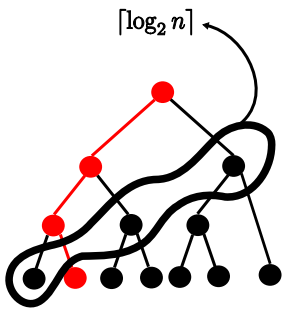


Con esto tenemos que la cota de $\lceil \frac{3}{2}n \rceil - 2$ comparaciones es ajustada, que nuestro algoritmo inicial no era óptimo, y que usamos el modelo de la cota inferior para ayudarnos a encontrar un algoritmo óptimo en términos del número de comparaciones.

1.1.4. Máximo y segundo máximo de un arreglo

Supongamos que deseamos encontrar el máximo y el segundo máximo elemento de $A[1, n]$. Una solución simple es encontrar el máximo y luego volver a encontrar el máximo entre los elementos restantes. Esto nuevamente cuesta $2n-3$ comparaciones. ¿Será óptimo? ¿Será que este problema es intrínsecamente más difícil que el de encontrar el máximo y el mínimo?

La analogía con el torneo de tenis nos sugiere una forma mucho mejor de resolver este problema. En un torneo de tenis, el segundo mejor debe haber jugado contra el primero, y sólo contra éste puede haber perdido. Como el primero realizó (y ganó) $\lceil \log_2 n \rceil$ partidas, hay sólo $\lceil \log_2 n \rceil$ candidatos para el segundo puesto. Una vez realizado el torneo de tenis para encontrar el máximo, podemos encontrar el segundo máximo entre los que perdieron contra el máximo usando $\lceil \log_2 n \rceil - 1$ comparaciones. El costo total es entonces $n + \lceil \log_2 n \rceil - 2$, lo que muestra que este problema es en realidad más fácil que el de encontrar el máximo y el mínimo, pues aquél requiere de $\lceil \frac{3}{2} n \rceil - 2$ comparaciones.



La pregunta natural es si nuestro algoritmo es óptimo, o el problema se puede resolver aún mejor.

Cota inferior incorrecta. Intentemos reusar el modelo de la tabla, con los siguientes conjuntos:

- a es el cardinal del conjunto A de los elementos que nunca han sido comparados;
- b es el cardinal del conjunto B de los elementos que se han comparado alguna vez y han ganado (han resultado mayores) en todas sus comparaciones;
- c es el cardinal del conjunto C de los elementos que han perdido (han resultado menores) exactamente una vez; y
- d es el cardinal del conjunto D de los elementos que han perdido más de una vez.

El estado inicial es $(n, 0, 0, 0)$ y el final debe ser $(0, 1, 1, n-2)$. La tabla es como sigue:

	A	B	C	D
A	$(a-2, b+1, c+1, d)$	$(a-1, b, c+1, d)$	~~$(a-1, b+1, c-1, d+1)$~~ $(a-1, b, c+1, d)$	$(a-1, b+1, c, d)$ $(a-1, b, c+1, d)$
B		$(a, b-1, c+1, d)$	$(a, b, c-1, d+1)$ $(a, b-1, c+1, d)$	(a, b, c, d) $(a, b-1, c+1, d)$
C			$(a, b, c-1, d+1)$	(a, b, c, d) $(a, b, c-1, d+1)$
D				(a, b, c, d)

Tal como en el caso del mínimo y máximo, obtenemos una cota inferior de $\lceil \frac{3}{2}n \rceil - 2$. ¡Pero esto no puede ser, ya tenemos una cota superior menor! ¿Qué ha ocurrido?

Lo que ha ocurrido es que nos hemos equivocado al suponer que es necesario llegar al estado $(0, 1, 1, n-2)$ para poder responder. En el torneo de tenis, casi la mitad de los jugadores juega un solo partido y queda descartada como primero o segundo, sin necesidad de haber perdido dos veces. La razón es la transitividad: si se pierde contra alguien que no es el mejor, no se puede ser el segundo mejor. Es decir, el algoritmo infiere cosas por transitividad, sin hacer comparaciones directas. Incluimos este ejemplo para mostrar que debe tenerse cuidado al aplicar esta técnica, asegurándose de que realmente es necesario llegar al estado final para poder responder correctamente.

Cota inferior correcta. Digamos que en un algoritmo que encuentra el máximo hay m elementos que se comparan directamente (y pierden) contra quien finalmente resulta ser el máximo. El segundo máximo es entonces el mayor de estos m candidatos (el segundo máximo debe haberse comparado contra el máximo, pues si no, ganó todas sus comparaciones y el adversario podría hacerlo arbitrariamente grande, incluso mayor que quien el algoritmo entrega como el máximo).

Consideremos de nuevo el modelo del grafo que se conecta. Si quitamos al nodo del máximo y a las m aristas que lo conectan con los candidatos a segundo máximo, el grafo debe aún resultar conexo para poder determinar el segundo máximo. De no ser así, existen dos componentes conexas que se unían sólo pasando por el máximo, y el paso por el máximo no sirve para determinar en cuál de las dos componentes está el segundo máximo.

Por lo tanto el grafo debe tener al menos $n + m - 2$ aristas, y se necesitan al menos $n + m - 2$ comparaciones para encontrar el máximo y el segundo máximo ($n - 1$ para el primero y $m - 1$ para el segundo). Mostraremos que un adversario puede conseguir que $m = \lceil \log_2 n \rceil$.

Consideremos el siguiente modelo para la cota inferior. Se asocia un peso $w(i)$ a cada celda $A[i]$, inicialmente $w(i) = 1$. Cuando un elemento $A[i]$ pierda una comparación, su $w(i)$ pasará a ser cero. Por lo tanto, para entregar el máximo correctamente, se requiere que haya un único $w(k) > 0$ (donde $A[k]$ será entonces el máximo).

Ahora diseñemos un adversario adecuado. Cuando el algoritmo compara $A[i]$ con $A[j]$, hay tres casos:

1. Si $w(i) > w(j)$, el adversario responde que $A[i] > A[j]$. Esto es consistente porque $A[i]$ no ha perdido ninguna comparación. Asimismo, el adversario actualiza $w(i) \leftarrow w(i) + w(j)$ y $w(j) \leftarrow 0$. Este caso incluye el $w(i) < w(j)$, mediante intercambiar i y j.

2. Si $w(i) = w(j) > 0$, el adversario se comporta como en el caso anterior, eligiendo arbitrariamente quién es i y quién j.

3. En otro caso, el adversario da cualquier respuesta que sea consistente con las anteriores (es decir, si de las comparaciones pasadas se puede deducir el resultado de esta comparación, ese resultado debe mantenerse). En este caso, no se actualizan las w.

Puede verse que este adversario agrega un par de invariantes más al modelo: (1) todas las w suman siempre n, y (2) cuando un $w(i)$ crece, a lo sumo se duplica. Eso implica que, para cuando el algoritmo puede responder correctamente que $A[k]$ es el máximo, vale que $w(k) = n$, y como llegamos de $w(k) = 1$ a $w(k) = n$ a lo sumo duplicándolo en cada comparación, el elemento $A[k]$ debe haberse comparado directamente al menos $\lceil \log_2 n \rceil$ veces.

Note que la cota inferior de $n - 1$ comparaciones para el máximo no requiere que el adversario responda de alguna manera especial en las comparaciones, por lo que podemos usar en particular este adversario para garantizar que, además de las $n - 1$ comparaciones para encontrar el máximo, se requerirán otras $\lceil \log_2 n \rceil - 1$ para el segundo máximo.

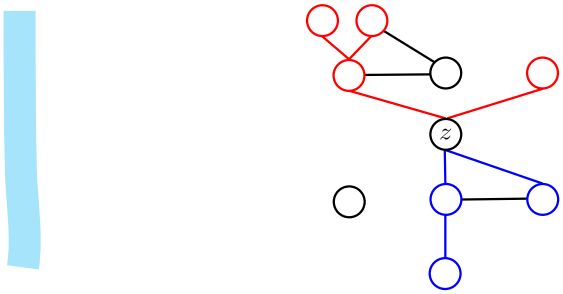
Finalmente, note que ningún algoritmo puede decir cuál es el segundo máximo si no sabe cuál es el máximo, pues eso significa que el segundo máximo propuesto no ha perdido ninguna comparación, y el adversario podría hacer que el segundo máximo propuesto fuera tan grande como quisiera. Por lo tanto, encontrar el segundo máximo es equivalente en dificultad a encontrar el primer y segundo máximo.

1.1.5. Mediana de un arreglo

Encontrar la mediana z de un arreglo $A[1, n]$ (con n impar) es un problema para el cual no se conoce el número exacto de comparaciones. Se conoce una cota inferior de $(2 + 2^{-50})n$ y una cota superior de $2{,}95n$. Mostraremos una cota inferior relativamente sencilla de $\frac{3(n-1)}{2}$ comparaciones. Hablaremos de la mediana z que entregará el algoritmo aunque éste no la conozca hasta el final.

Consideraremos dos tipos de comparaciones, cruciales y no cruciales. Conceptualmente, una comparación resulta *crucial* para un elemento x si es la que nos permite conocer la relación entre x y z. Más precisamente, consideremos la historia de las comparaciones que realizó el algoritmo para determinar z, y definamos un grafo con un nodo por elemento. Dibujemos una arista entre z y los elementos x que se compararon directamente contra z, roja si $x > z$ y azul si $x < z$. También pintemos a x de rojo o azul, respectivamente. Para todo nodo rojo x, dibujemos aristas rojas hacia elementos $y > x$ que aún no tengan color y se hayan comparado directamente contra x. Para todo nodo azul x, dibujemos aristas azules hacia elementos $y < x$ que aún no tengan color y se hayan comparado directamente contra

x. En ambos casos, pintemos a y de rojo o azul, respectivamente. Continuemos así hasta pintar todas las aristas y elementos posibles. Todas las aristas pintadas corresponden a las comparaciones cruciales.



El grafo formado por las aristas rojas y azules no tiene ciclos. Si no resulta conexo, el algoritmo no puede conocer la mediana, pues implica que existe un elemento x no pintado, por lo cual el algoritmo nunca hizo una comparación que le permitiera determinar si $x < z$ ó $x > z$. El adversario puede entonces decidir si $x < z$ ó $x > z$, haciendo que la respuesta z sea incorrecta. Por lo tanto, se necesitan al menos $n - 1$ comparaciones cruciales.

Mostraremos que, además, el algoritmo debe haber realizado al menos $\frac{n-1}{2}$ comparaciones no cruciales, cuyas aristas no están en el grafo porque resultan ser $x < y$ para un $y > z$, o bien $x > y$ para un $y < z$. Para esto, consideremos un adversario que responde a las comparaciones del algoritmo mediante asignarles valor a los elementos cuando los ve por primera vez. Antes de ello, determinará un valor z para quien será la mediana, sin asignárselo a ningún elemento en particular. Modelaremos el avance del conocimiento del algoritmo partiendo los elementos en tres conjuntos:

- a es el cardinal del conjunto A de los elementos que nunca han sido comparados (no confundir con el arreglo A);
- b es el cardinal del conjunto B de los elementos que se han comparado alguna vez y se les asignó un valor mayor a z; y
- c es el cardinal del conjunto C de los elementos que se han comparado alguna vez y se les asignó un valor menor a z.

El algoritmo no conoce la mediana hasta el final (es decir, no sabe qué tipo de comparación está realizando). Cuando se comparen dos elementos de A, el adversario les dará a uno un valor mayor y a otro un valor menor que z. Cuando se compare un elemento de A con uno de B, le asignará al de A un valor menor a z. Cuando se compare un elemento de A con uno de C, le asignará al de A un valor mayor a z. Note que en estos tres casos, la comparación resultará no ser crucial. Cuando se comparen elementos de B ó C, responderá según los valores que ya ha asignado (estas comparaciones podrían ser cruciales).

Con estas decisiones del adversario, la siguiente tabla muestra cómo progresa el estado (a, b, c) según los elementos que se comparan:

	A	B	C
A	$(a-2, b+1, c+1)$	$(a-1, b, c+1)$	$(a-1, b+1, c)$
B		(a, b, c)	(a, b, c)
C			(a, b, c)

Si llegamos a $b = \frac{n-1}{2}$, el adversario asignará a C todos los elementos aún no comparados (es decir, les dará valores menores a z), salvo uno que se reservará para asignarle el valor z. Similarmente, si llegamos a $c = \frac{n-1}{2}$, el adversario asignará a B todos los elementos aún no comparados menos uno. Con ello, z resultará ser la mediana, como era el plan del adversario.

De la tabla se deduce que, como partimos de $(n, 0, 0)$ y continuamos hasta que b ó c son $\frac{n-1}{2}$, necesitamos al menos ese número de comparaciones de la primera fila de la tabla, todas las cuales son no cruciales. Se deduce entonces la cota inferior de $\frac{3(n-1)}{2}$ comparaciones para encontrar la mediana.

Se puede usar el mismo razonamiento para demostrar que encontrar el k-ésimo elemento de un conjunto requiere $n + \min(k, n-k) - 2$ comparaciones, si bien esta cota no es ajustada.

1.2. Teoría de la Información

La *Teoría de la Información* es una disciplina que estudia la cantidad mínima de bits necesaria para representar un mensaje u objeto. Es decir, establece cotas inferiores para la cantidad de bits que debe emitir cualquier programa que represente un objeto, por comprimida que sea esta representación.

Es posible entonces obtener cotas inferiores a la cantidad de comparaciones que realiza un algoritmo sobre un input, si a partir de esas comparaciones podemos reconstruir ese input. En ese caso, el algoritmo podría considerarse potencialmente como un mecanismo de compresión, y la cantidad mínima de bits que debe emitir cualquier compresor se convierte en la cantidad mínima de comparaciones que debe realizar cualquier algoritmo. Debe notarse que este tipo de cotas aplica casi exclusivamente a algoritmos que deban proceder por comparaciones, pero a cambio puede establecer cotas tanto de peor caso como de caso promedio.

1.2.1. Cotas de peor caso

Para cotas de peor caso, la cota inferior es simplemente el logaritmo (base 2) del número total de inputs posibles. Si el conjunto de inputs posibles es U, entonces no es posible representar a todos sus elementos usando siempre menos de $\log_2 |U|$ bits. La razón es que el número total de descripciones que usan menos de $\log_2 |U|$ bits es $\sum_{\ell=0}^{\log_2 |U|-1} 2^\ell = |U| - 1$, es decir, no hay suficientes descripciones distintas para todos los elementos de U. Por lo tanto, cualquier algoritmo a través de cuyas comparaciones se pueda identificar el input requiere

$\log_2 |U|$ comparaciones en el peor caso. Note que, para hablar de bits, las comparaciones deben ser binarias, es decir, con dos resultados posibles (por ejemplo, $\leq$ y $>$).

Volvamos al juego de las 20 preguntas. Si A conoce a más de 2^{20} personajes distintos, entonces B no tiene suficientes preguntas para poder ganar siempre, pues no bastan 20 "bits" sí/no para identificar a todos los posibles personajes. Dicho de otro modo, para cualquier estrategia que B tenga, siempre existe un subconjunto de al menos dos personajes que no se llegan a distinguir con las primeras 20 preguntas.

Por otro lado, si A conoce no más de 2^{20} personajes, entonces B puede ganar siempre si busca preguntas *balanceadas*, es decir que a cada paso dividan el conjunto de personajes posibles a la mitad. Es por eso que preguntas como "¿tiene pelo negro?" o "¿nació en este continente?" son mejores que "¿se trata de Napoleón?" como primeras preguntas.

1.2.2. Búsqueda en un arreglo ordenado

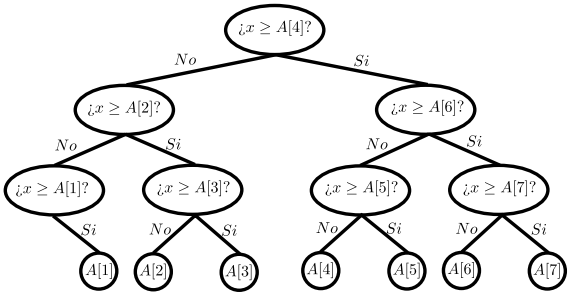
Una alternativa al método del adversario usado en la búsqueda en un arreglo ordenado es la siguiente. Si un algoritmo toma los objetos de búsqueda como cajas negras y se basa únicamente en comparaciones para tomar sus decisiones, entonces para cada búsqueda de un elemento $A[i]$ distinto deberá obtener una secuencia distinta de resultados a sus comparaciones (las comparaciones que realiza también pueden depender del resultado de comparaciones anteriores). Si existen dos elementos $A[i]$ y $A[j]$ para los cuales el algoritmo obtiene la misma secuencia de resultados, entonces es que realiza las mismas comparaciones y responde lo mismo a ambas búsquedas, lo cual sería incorrecto.

Eso significa que el algoritmo se puede convertir en un codificador para los valores en $[1, n]$. Creo un arreglo ordenado cualquiera $A[1, n]$, y para codificar i pido al algoritmo que busque $A[i]$. Tomo nota de los resultados de las comparaciones que va realizando el algoritmo y los codifico como una secuencia de bits. Para obtener i a partir de ese código (es decir, para "descomprimir" el valor de i), simulo el algoritmo, y cuando llega el momento de una comparación, veo qué pasaría si el resultado de la comparación se corresponde con el siguiente bit de la codificación.

Dado que un algoritmo de búsqueda permite codificar cada elemento de $[1, n]$ mediante las comparaciones que realiza, se deduce que en el peor caso debe realizar al menos $\log_2 n$ comparaciones. También esto sugiere que, para llegar a ese peor caso, el algoritmo debe procurar que cada comparación reduzca a la mitad el número de inputs posibles, lo que nuevamente nos lleva a la búsqueda binaria.

1.2.3. Ordenar un arreglo

Consideremos el problema de ordenar $A[1, n]$. Ordenar implica aplicar una permutación π al rango $[1, n]$ de modo que A quede ordenado. Esto significa que la permutación original que traían los elementos de A, ρ, es la inversa de π. Para cada ρ posible, un algoritmo correcto de ordenamiento debe aplicar un conjunto distinto de operaciones $\pi = \rho^{-1}$. Un



algoritmo que procede únicamente por comparaciones puede entonces utilizarse para codificar la permutación π (o ρ) de la misma manera que en el ejemplo anterior: los resultados de las comparaciones que va realizando (ignorando las modificaciones que hace al arreglo o cualquier otra operación) deben ser distintas para cada input posible ρ.

Dado que existen $n!$ posibles permutaciones en las que A puede presentarse, el algoritmo necesita realizar al menos $\log_2(n!)$ comparaciones para poder realizar un conjunto de acciones distinto para cada una de ellas. Por la aproximación de Stirling, $\log_2(n!) = n\log_2 n - O(n)$. Dicho más gruesamente, para ordenar por comparaciones se necesita tiempo $\Omega(n \log n)$.

1.2.4. Unir dos listas ordenadas

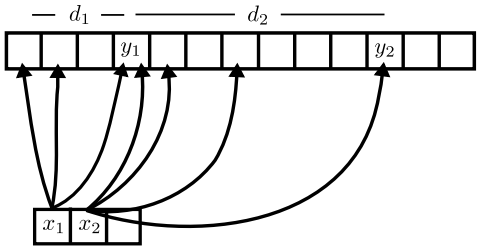
¿Cuántas comparaciones se necesitan para unir dos listas ordenadas, de largo n y m, con $m < n$? Una cota inferior está dada por el número de formas en que los elementos de una lista se pueden insertar en la otra, $\log_2 \binom{m+n}{m}$. Esto ocurre porque todo algoritmo que haga la unión debe realizar acciones distintas para cada una de las $\binom{m+n}{m}$ formas en que puede presentarse el input, y por lo tanto las respuestas a sus comparaciones pueden usarse para codificar ese aspecto del input.

Note que, si $p \geq q$,

$$\binom{p}{q} = \prod_{i=0}^{q-1} \frac{p-i}{q-i} \geq \prod_{i=0}^{q-1} \frac{p}{q} = \left(\frac{p}{q}\right)^q$$

debido a que $\frac{p-i}{q-i} \geq \frac{p}{q}$ pues $(p-i)q = pq - iq \geq pq - ip = p(q-i)$. De esto obtenemos la cota inferior $\log_2 \binom{m+n}{m} \geq \log_2((\frac{m+n}{m})^m) = m\log_2(1+\frac{n}{m})$.

Asintóticamente, esta cota inferior es $\Omega(m \log \frac{n}{m})$ si $m = o(n)$ y $\Omega(m)$ si no. Un algoritmo que alcanza esta cota inferior asintótica es el que toma la lista más corta y avanza en la más larga buscándolo mediante búsqueda exponencial (es decir, comparando el elemento en las posiciones 1, 2, 4, ... hacia adelante hasta pasarse y luego completando con búsqueda binaria). Para cada elemento x_i de la lista más corta, sea y_i el primer elemento $\geq x_i$ de la lista más larga. Si la distancia entre y_i e y_{i-1} es de d_i posiciones (en la lista más larga), entonces el costo total será $O(\sum_i \log d_i)$.



Como $\sum_i d_i \leq n$, el peor caso se da cuando son todos $d_i = \frac{n}{m}$ (por la desigualdad de Jensen), en cuyo caso el algoritmo toma tiempo $O(m(1 + \log \frac{n}{m}))$.

Cuando $m = n$, la cota inferior es $\log_2 \binom{2n}{n} \geq 2n - O(\log n)$ (por la aproximación de Stirling), cuyo término principal exacto se alcanza con el método típico de recorrer ambas listas secuencialmente e ir tomando el menor $(2n - 1$ comparaciones).

1.2.5. Cotas de caso promedio

El Teorema de Shannon establece que, si los elementos de $U = \{u_1, \ldots, u_n\}$ se presentan con probabilidad p_i para u_i (con $\sum p_i = 1$), entonces ningún compresor puede, en promedio, utilizar menos de

$$H = \sum_i p_i \log_2 \frac{1}{p_i}$$

bits para codificar un elemento de U. Este valor se llama la *entropía* del conjunto de probabilidades. La entropía nos da una herramienta para establecer cotas inferiores en el caso promedio, lo que no se da con la estrategia del adversario.

Note que, si todas las $p_i = \frac{1}{|U|}$, entonces la entropía resulta ser $H = \log_2 |U|$, llegando a su valor máximo. Como esto coincide con el valor del peor caso, resulta que la cota inferior de peor caso de un algoritmo (demostrada con esta técnica) también es la cota inferior de caso promedio si los inputs se presentan todos con la misma probabilidad. En particular, de los ejemplos anteriores, tenemos que la cota de $\log_2 n$ comparaciones para buscar en un arreglo ordenado o de $n \log_2 n - O(n)$ para ordenar también se aplican al caso promedio del mejor algoritmo posible, si suponemos que los elementos de A se buscan con la misma probabilidad o que todos los reordenamientos de entrada de A son igualmente probables, respectivamente.

Por otro lado, si resulta que tenemos que buscar en un arreglo y ciertos elementos se buscan con mayor probabilidad que otros, entonces podemos romper la cota de $\log_2 n$ comparaciones en promedio, y llegar a la entropía H. Pero, ¿cómo hacerlo?

1.2.6. Árboles de búsqueda óptimos

Nuestro problema se puede describir como: encontrar un árbol de búsqueda para A donde la profundidad promedio de una hoja sea H. Note que cada árbol de búsqueda que podamos diseñar sobre A corresponde a un algoritmo, o a una estrategia, de búsqueda, mientras

que cada búsqueda en concreto es un camino de comparaciones desde la raíz hasta la hoja correspondiente.

Algoritmos de Huffman y Hu-Tucker. El problema de, dado un conjunto de probabilidades p_i, encontrar el árbol binario que minimice $\sum p_i \ell_i$, donde ℓ_i es la profundidad de la hoja de probabilidad p_i, es conocido en compresión. Una estrategia que entrega el árbol óptimo es el algoritmo de Huffman, que puede describirse de la siguiente forma.

1. Crear un bosque con n árboles, cada uno consistente de un único nodo (u_i) de peso $w = p_i$.
2. Tomar los dos árboles T_1 y T_2 del bosque con menor peso, sean w_1 y w_2 esos pesos, y colocarlos como hijos izquierdo y derecho de un nuevo nodo.
3. Sacar T_1 y T_2 del bosque, y agregar el nuevo árbol, con peso $w_1 + w_2$.
4. Volver al punto 2 a menos que el bosque contenga un solo árbol.

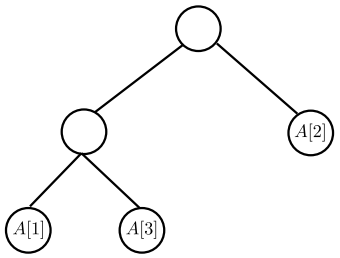
El árbol final que entrega este algoritmo tiene la propiedad de que minimiza $L = \sum p_i \ell_i$, y además puede demostrarse que $H \leq L < H + 1$, es decir, la profundidad promedio de las hojas es menos que la entropía más 1.

Antes de continuar, ¿cuánto tiempo requiere este algoritmo? Es fácil ver que se puede hacer en tiempo $O(n \log n)$, mediante almacenar los pesos de los árboles en una cola de prioridad, implementada por ejemplo con un heap. Se crea con n elementos en tiempo $O(n)$ y luego se realizan $2n$ extracciones y n reinserciones. ¿Puede hacerse mejor? No se sabe en general, pero si los elementos ya vienen ordenados por probabilidad, entonces puede hacerse en tiempo $O(n)$. Ponga los n nodos iniciales en una cola, y prepare otra cola vacía de tamaño $n - 1$ para encolar los árboles nuevos que va produciendo. Esta nueva cola también estará ordenada porque los pesos de los árboles que se van generando son siempre crecientes. Entonces, cuando tenga que extraer los dos nodos de peso mínimo, extráigalos de cualquiera de las dos colas (uno de cada una o los dos de una cola, según dónde estén los menores).

De todos modos, ¿el árbol de Huffman sirve como árbol de búsqueda? Supongamos $A[1, 3]$ con probabilidades $p_1 = p_3 = \frac{1}{4}$ y $p_2 = \frac{1}{2}$. El árbol de Huffman tendrá un hijo de la raíz para la hoja $A[2]$ y el otro será un nodo interno, con hijos hoja para $A[1]$ y $A[3]$. No es posible hacer una comparación tipo "¿$A[i] \leq x$?" para separar $A[2]$ de $A[1]$ y $A[3]$. La razón es que el algoritmo de Huffman desordena las hojas, por lo cual no necesariamente entrega un árbol de búsqueda.

Existe un algoritmo que entrega el mejor árbol posible sin desordenar las hojas, toma tiempo $O(n \log n)$ y garantiza que $H \leq L < H + 2$. Se llama algoritmo de Hu-Tucker. Esto significa que existe un algoritmo de búsqueda para arreglos ordenados que realiza en promedio menos de 2 comparaciones por encima de la cota inferior de Teoría de la Información.

En realidad, podemos demostrar que la cota H no es ajustada, y que lo más cercano ajustado es, precisamente, $H + 2$. Considere $A[1, 3]$ con probabilidades $p_1 = p_3 = \epsilon$ y $p_2 =$



$1-2\epsilon$, para $\epsilon > 0$ tan pequeño como se quiera. Entonces vale $H = 2\epsilon \log_2 \frac{1}{\epsilon} + (1-2\epsilon) \log_2 \frac{1}{1-2\epsilon}$, el cual tiende a cero cuando $\epsilon \to 0$. Sin embargo, el mejor árbol de búsqueda posible tiene al nodo $A[2]$ a profundidad 2 y un costo promedio de búsqueda $2(1-2\epsilon)+3\epsilon$, que tiende a 2 cuando $\epsilon \to 0$.

No describiremos el algoritmo de Hu-Tucker en este apunte, por ser demasiado complicado y alejarse demasiado de nuestro tema principal. En cambio, veremos un algoritmo más costoso que resuelve un problema más general.

Construcción general. Consideremos un modelo distinto, en que procedemos por comparaciones pero éstas pueden ser $<$, $=$, ó $>$. Esto significa que podemos detenernos en un nodo interno del árbol de búsqueda si encontramos el elemento que buscamos. En este caso, en vez de comparaciones, contaremos la cantidad de accesos a A.

Para generalizar, supondremos que las búsquedas pueden ser *exitosas* o *infructuosas*. Una búsqueda exitosa encuentra lo que busca en el elemento $A[i]$ del arreglo, el cual se busca con probabilidad p_i. Una búsqueda infructuosa no encuentra lo que busca, sino que determina que debería estar entre los elementos $A[i]$ y $A[i+1]$ del arreglo, suponiendo implícitamente que $A[0] = -\infty$ y $A[n+1] = +\infty$. Diremos que esta búsqueda ocurre con probabilidad q_i. Tenemos entonces $\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1$.

El árbol de búsqueda se puede ver entonces como un árbol de n nodos donde los nodos internos representan la lectura de un elemento, y entendemos que, si buscamos ese elemento, la búsqueda termina allí. Los $n+1$ punteros a nulo que salen de las hojas son las búsquedas infructuosas, cuyos resultados se deducen sólo al final del camino hacia una hoja. El costo de una búsqueda exitosa, medido en cantidad de accesos a A, es la profundidad del nodo interno correspondiente, y el de una búsqueda infructuosa es la profundidad del nodo hoja correspondiente (y es igual a la búsqueda exitosa para ese nodo). Nuestro modelo anterior se puede simular suponiendo un arreglo de tamaño $n-1$, cuyas $n-1$ búsquedas exitosas tienen probabilidad $p_i = 0$, y sólo existen búsquedas infructuosas (que terminan en hojas).

Podemos construir el árbol binario óptimo para este problema mediante programación dinámica. Llamemos

$$P_{i,j} \;=\; q_{i-1} + p_i + q_i + \ldots + p_j + q_j$$

a la probabilidad de que la búsqueda recaiga sobre el rango $A[i,j]$, incluyendo las búsquedas infructuosas en sus extremos. Entonces, el costo óptimo para buscar en $A[i,j]$ se puede

encontrar como $C_{i,i-1} = 0$ (arreglo vacío, búsqueda infructuosa), $C_{i,i} = 1$ (arreglo de 1 elemento, búsqueda exitosa o infructuosa), y, para $i < j$,

$$C_{i,j} = 1 + \min_{i \le k \le j} \frac{P_{i,k-1}}{P_{i,j}} \cdot C_{i,k-1} + \frac{P_{k+1,j}}{P_{i,j}} \cdot C_{k+1,j},$$

donde k representa la raíz que elijamos para este subárbol de búsqueda, es decir, el elemento de A sobre el que haremos la primera comparación. Debemos registrar dónde queda esta raíz, para luego poder reconstruir el árbol:

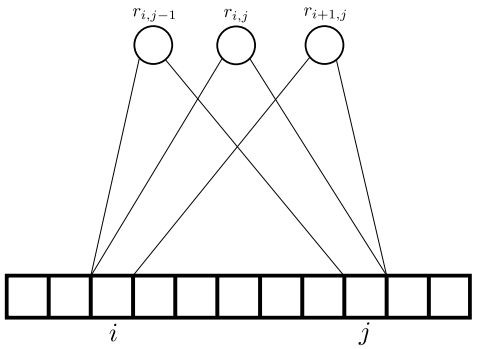
$$r(i,j) = \arg\min_{i \le k \le j} \frac{P_{i,k-1}}{P_{i,j}} \cdot C_{i,k-1} + \frac{P_{k+1,j}}{P_{i,j}} \cdot C_{k+1,j}.$$

Una vez calculadas estas dos matrices (por ejemplo por diagonales, partiendo de la diagonal principal y la que está bajo ella, y progresando hasta la diagonal de largo 1 de la celda $C_{1,n}$), tendremos en $C_{1,n}$ el costo promedio de la búsqueda usando la mejor estrategia posible, y en $r(1,n)$ la raíz que debemos usar para esa estrategia (es decir, debemos partir examinando la celda $A[r(1,n)]$). El hijo izquierdo debe tener raíz $r(1, r(1,n)-1)$ y el derecho $r(r(1,n)+1, n)$, y así sucesivamente.

Note que, si bien el árbol final requiere de solamente $O(n)$ espacio, necesitamos $O(n^2)$ espacio para encontrarlo usando programación dinámica. El tiempo para calcular la matriz P es también $O(n^2)$, pues cada celda puede calcularse con $P_{i,i-1} = q_{i-1}$ y luego $P_{i,j} = P_{i,j-1} + p_j + q_j$ para todo $j \ge i$. En cambio, las matrices C y r requieren tiempo $O(n^3)$, pues deben considerarse todos los $j - i + 1$ posibles valores de k.

Cuando los costos de acceso son iguales (unitarios, en nuestro modelo), es posible calcular C y r en tiempo $O(n^2)$, mediante usar la importante propiedad (que no demostraremos en este apunte) de que

$$r(i, j-1) \le r(i,j) \le r(i+1, j).$$



Eso significa que podemos reescribir la fórmula para calcular las celdas $C_{i,j}$ como

$$C_{i,j} = 1 + \min_{r(i,j-1) \le k \le r(i+1,j)} \frac{P_{i,k-1}}{P_{i,j}} \cdot C_{i,k-1} + \frac{P_{k+1,j}}{P_{i,j}} \cdot C_{k+1,j},$$

y similarmente para $r(i,j)$. Para ver que esto reduce el costo total a $O(n^2)$, consideremos el costo a lo largo de una diagonal d de la matriz, donde $j = i + d$. Entonces, calcular la celda $C_{i,j}$ cuesta

$$r(i+1,j) - r(i,j-1) + 1 \;=\; r(i+1,i+d) - r(i,i+d-1) + 1.$$

Calcular la siguiente celda de la diagonal cuesta

$$r(i+2,j+1) - r(i+1,j) + 1 \;=\; r(i+2,i+d+1) - r(i+1,i+d) + 1.$$

Y la siguiente cuesta

$$r(i+3,j+2) - r(i+2,j+1) + 1 \;=\; r(i+3,i+d+2) - r(i+2,i+d+1) + 1.$$

Si sumamos estos costos, puede verse que cada segundo término se cancela con el primer término anterior. Por lo tanto la suma es telescópica, y a lo largo de la diagonal d el costo es a lo sumo $r(n-d,n) - r(1,d+1) + n - d \leq 2n$. A lo largo de las n diagonales, el costo suma $O(n^2)$. Este es otro pequeño ejemplo de análisis amortizado: una celda puede demorar $O(n)$ en calcularse, pero vemos que las $O(n^2)$ celdas no requieren más de $O(n^2)$ operaciones.

1.3. Reducciones

Las reducciones se usan en el diseño de algoritmos para encontrar una solución a un problema desconocido mediante *reducirlo* a uno conocido (por ejemplo, reducir el problema de encontrar elementos repetidos en un arreglo al de ordenarlos y hacer una pasada buscando repetidos consecutivos). En cambio, en la teoría de NP-completitud, se demuestra que un problema es NP-completo mediante reducir un problema NP-completo conocido a él. Es decir, operamos en la dirección contraria: el problema conocido se reduce al problema desconocido.

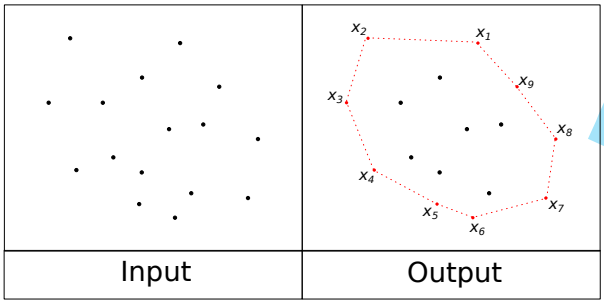
En el caso de cotas inferiores, la idea es similar a la de NP-completitud. Supongamos que tenemos un problema P para el que queremos establecer una cota inferior, y conocemos un problema Q con una determinada cota inferior $\Omega(C(n))$. Si podemos transformar un input de Q en uno de P en tiempo $o(C(n))$, resolver P en el input transformado, y luego transformar el output de P en el de Q también en tiempo $o(C(n))$, entonces $\Omega(C(n))$ es también una cota inferior para el problema P. De no ser así, las transformaciones nos darían una solución a Q de tiempo $o(C(n))$, lo que es imposible.

Esta técnica es general y puede usarse para peor caso y caso promedio, si bien generalmente se usa para establecer órdenes de magnitud y no cantidades exactas de operaciones.

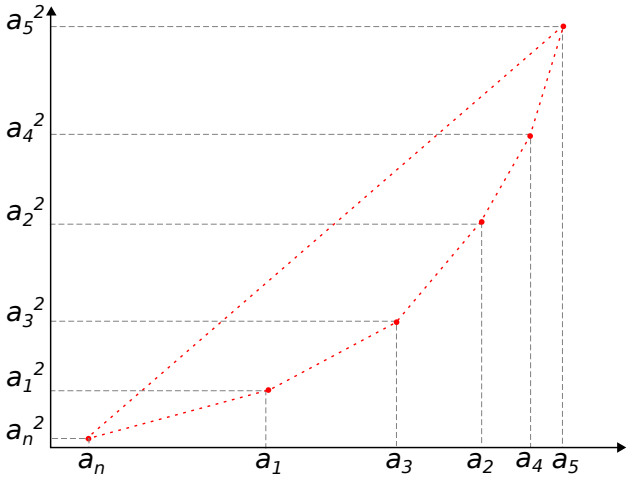
1.3.1. Cápsula convexa

El problema de la *cápsula convexa* es el de, dados n puntos en el plano, encontrar el menor polígono convexo que los contiene. Es fácil ver que los vértices de este polígono deben

ser puntos del input, por lo cual el output del problema se pide en la forma de la secuencia de puntos que se obtienen al recorrer el perímetro del polígono en sentido antihorario, partiendo desde algún punto.



Si consideramos que las coordenadas son números reales, sobre los que únicamente podemos hacer operaciones matemáticas y comparaciones, entonces podemos mostrar que este problema es $\Omega(n \log n)$ mediante reducir el problema de ordenar al de la cápsula convexa. Para ordenar $A = \{a_1, \ldots, a_n\}$, calculamos n puntos $\{(a_1, a_1^2), \ldots, (a_n, a_n^2)\}$.



Es fácil ver que los n puntos están distribuidos en una parábola, por lo cual todos formarán parte de la cápsula convexa. Más aún, un listado en sentido antihorario de los puntos que parta del mínimo recorre, en sus primeras coordenadas, al conjunto X en orden de menor a mayor. Una pasada simple sobre el output de la cápsula convexa nos permite detectar el menor elemento y a partir de él listar a todos en orden. Como la transformación del input y del output nos cuesta $\Theta(n) = o(n \log n)$, tenemos que $\Omega(n \log n)$ es una cota inferior al problema de calcular la cápsula convexa.

En realidad esta cota inferior puede refinarse si introducimos otras variables. Por ejemplo, si h es el número de puntos en el output, existen algoritmos que resuelven el problema en tiempo $O(n \log h)$. Sin embargo, esto es todavía $O(n \log n)$ en el peor caso.

1.3.2. Colas de prioridad

Esta técnica también nos permite establecer cotas inferiores al costo de realizar una secuencia de operaciones sobre una estructura de datos.

Por ejemplo, la implementación basada en un *heap* obtiene tiempos $O(\log n)$ para las operaciones de insertar y extraer el mínimo en una cola de prioridad. Existen implementaciones, como las colas de Fibonacci (que mencionaremos en el capítulo de análisis amortizado) donde la inserción se puede hacer en tiempo $O(1)$, pero la extracción del mínimo aún cuesta $O(\log n)$. Incluso se puede crear un heap de n elementos en tiempo $O(n)$. Nos preguntamos si existirá alguna implementación de colas de prioridad donde se pueda insertar un elemento en tiempo $O(1)$ y extraer el mínimo en tiempo $O(1)$.

No es difícil ver que esto es imposible si se procede por comparaciones. Si lo fuera, podríamos reducir el problema de ordenar n elementos al de insertarlos en una cola de prioridad vacía y extraer el mínimo, luego el mínimo de lo que queda, y así sucesivamente hasta obtener el arreglo ordenado. Si se pudiera extraer el mínimo en tiempo $o(\log n)$, podríamos ordenar en tiempo $o(n \log n)$. Note que esto vale incluso en promedio, si las inserciones agregan los elementos en un orden uniformemente aleatorio.

1.3.3. 3SUM y puntos colineales

El concepto de reducción se utiliza también para hablar de cotas inferiores que están en función de otras cotas inferiores que no son conocidas, pero sí muy estudiadas. Por ejemplo, si podemos reducir la multiplicación de matrices de $n \times n$ a un cierto problema P, sabemos que P no es más fácil que multiplicar matrices. Antes de Strassen, podríamos haber pensado que la complejidad del problema era $\Theta(n^3)$. En 1969, Strassen mostró cómo multiplicar matrices en tiempo $O(n^{2,81})$, y luego en 1990 Coppersmith y Winograd lo redujeron a $O(n^{2,38})$ (analizado en 2014 por Le Gall). No se conoce una cota inferior general para el problema más allá de la obvia $\Omega(n^2)$. Un problema tan trabajado es útil en sí mismo como cota inferior: si podemos resolver P en menor tiempo, habremos encontrado un algoritmo para multiplicar matrices mejor que todos los conocidos. Decimos entonces que P es multiplicación-de-matrices-hard. Tal como en la NP-completitud, donde se dice que un problema es NP-hard, es una forma de decir cuán improbable se considera obtener un mejor resultado para resolver P (nota: ser NP-completo equivale a ser NP y ser NP-hard).

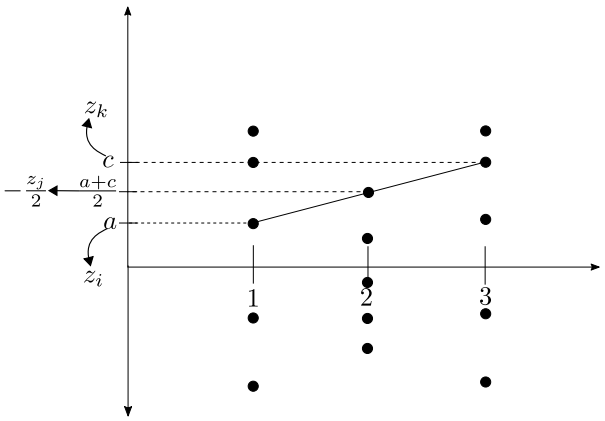
El problema *3SUM* es el de, dados n números reales $Z = \{z_1, \ldots, z_n\}$, encontrar tres que sumen cero (pueden repetirse números en la suma). Veamos cómo resolver este problema en tiempo $O(n^2)$. Supongamos que primero ordenamos los números en tiempo $O(n \log n)$. Luego, tomaremos cada número z_i y buscaremos dos números de Z que sumen $-z_i$. Para ello, progresaremos desde las dos puntas del arreglo ordenado, $m \leftarrow z_1$ y $M \leftarrow z_n$, con dos cursores. Si $m + M + z_i < 0$, moveremos el cursor de la izquierda hacia adelante, $m \leftarrow z_2$. En cambio, si $m + M + z_i > 0$, moveremos el cursor de la derecha hacia atrás, $M \leftarrow z_{n-1}$. En todo momento, el invariante es que los números que ya dejamos de considerar no pueden

formar parte de la solución, y se puede ver que se mantiene cuando movemos los cursores. Al final, en tiempo $O(n)$ encontramos un m y M adecuados (con lo cual resolvimos el problema con los números m, M y z_i), o los cursores se cruzan y debemos pasar a considerar otro número z_i. El tiempo total es $O(n^2)$.

Por mucho tiempo se sospechó que $\Theta(n^2)$ era la complejidad del problema, pero en 2014 Grunlund y Pettie encontraron una solución de tiempo $O(n^2/(\log n/\log\log n)^{2/3})$. Aún así, se sospecha que el problema es $\Omega(n^{2-o(1)})$ (es decir, $\Omega(n^{1,999\cdots})$ para cualquier cantidad finita de 9s). Como es un problema bastante estudiado, es interesante cuando se establece que otro problema es 3SUM-hard.

Consideremos el problema de, dados n puntos (x_i, y_i), encontrar *tres puntos colineales* que no estén en una línea vertical. Esta última restricción se pone por conveniencia para demostrar una cota inferior, pues ese subcaso es fácil de resolver.

Reduciremos 3SUM a este problema, para mostrar que es 3SUM-hard. Dados n números $Z = \{z_1, \ldots, z_n\}$, generaremos 3 puntos para cada z_i: $(1, z_i)$, $(2, -\frac{z_i}{2})$, y $(3, z_i)$. Veremos que este conjunto tiene 3 puntos colineales no en vertical sii Z tiene 3 números que suman 0. Dadas las coordenadas x_i elegidas, estos 3 puntos colineales deben ser de la forma $(1, a)$, $(2, b)$ y $(3, c)$, con $b = \frac{a+c}{2}$. Pero como $a = z_i$ para algún i, $b = -\frac{z_j}{2}$ para algún j, y $c = z_k$ para algún k, tenemos que $-\frac{z_j}{2} = \frac{z_i+z_k}{2}$, de lo que se deduce que $z_i + z_j + z_k = 0$. También puede verse que ocurre lo recíproco: si hay tres números que sumen cero, hay tres puntos colineales. Como la conversión cuesta sólo $O(n)$, tenemos que el problema es 3SUM-hard.



Finalmente, el problema de encontrar tres puntos colineales sin la restricción de que estén en vertical también es 3SUM-hard. Para que esto tenga sentido, sin embargo, aún debemos prohibir que los puntos sean iguales, pues el problema es trivial en ese caso. Si los tres puntos deben ser distintos, entonces podemos crear los puntos (z_i, z_i^3). Si aparecen tres puntos colineales distintos (a, a^3), (b, b^3) y (c, c^3), con $a < b < c$, entonces tenemos que $\frac{a^3-b^3}{a-b} = \frac{b^3-c^3}{b-c}$, es decir, $a^2 + ab + b^2 = b^2 + bc + c^2$, de donde tenemos $a^2 - c^2 = -b(a - c)$, o $(a + c)(a - c) = -b(a - c)$. Dividiendo por $a - c$ obtenemos $a + b + c = 0$.

1.4. Ficha Resumen

Técnicas:

- Estrategia del adversario
- Teoría de la Información
- Reducciones

Complejidad de problemas:

- Encontrar un elemento en arreglo desordenado: n accesos.
- Encontrar un elemento en arreglo ordenado: $\lceil \log_2 n \rceil$ accesos o comparaciones.
- Encontrar el máximo en un arreglo: $n-1$ comparaciones.
- Encontrar el mínimo y el máximo en un arreglo: $\lceil \frac{3}{2}n \rceil - 2$ comparaciones.
- Encontrar el máximo y segundo máximo en un arreglo: $n + \lceil \log_2 n \rceil - 2$ comparaciones.
- Encontrar la mediana en un arreglo: entre $(2+2^{50})n$ y $2{,}95n$ comparaciones (vimos cota inferior de $\lceil \frac{3(n-1)}{2} \rceil$).
- Ordenar un arreglo: $n \log_2 n - O(n)$ comparaciones, algoritmos como MergeSort hacen $n \log_2 n + O(n)$.
- Mergear dos listas de largo $m < n$: $\Theta(m \log \frac{n}{m})$, y $2n - O(\log n)$ si $m = n$.
- Buscar en un arreglo con entropía de probabilidades H: $H + 2$ comparaciones.
- Calcular la cápsula convexa: $\Theta(n \log n)$ operaciones y comparaciones sobre reales.
- Encontrar tres números que sumen cero (3SUM): $O(n^2/(\log n/\log\log n)^{2/3})$, se sospecha $\Omega(n^{2-o(1)})$.
- Encontrar tres puntos colineales: 3SUM-hard.

1.5. Material Suplementario

Lee et al. [LTCT05, sec. 2.3] dedican una sección a explicar la idea general de cotas inferiores y superiores de problemas. Levitin [Lev07, sec. 11.1] también dedica una sección a presentar la idea general, con los tres enfoques que consideramos en el apunte. Esta sección tiene varios ejemplos simples, incluyendo la cota para la unión de listas ordenadas. Algo más corta, Baase [Baa88, sec. 3.1] da una descripción general de la idea del adversario. Estas secciones son buenas guías iniciales, sorprendentemente no muy comunes en la literatura.

Baase [Baa88, sec. 3.2–3.4] describe un mecanismo similar al que expusimos para la cota inferior de encontrar el máximo y el mínimo, así como el problema del máximo y segundo máximo, y el problema de la mediana (describiendo también un algoritmo para esta última).

Mucho más popular en la literatura es la cota inferior de $\Omega(n \log n)$ para ordenar por comparaciones. Por ejemplo, la explican Aho et al. [AHU83, sec. 8.6], quienes incluso discuten el caso promedio (de una forma algo distinta al apunte). Manber [Man89, sec. 6.4.6], Cormen et al. [CLRS01, sec. 8.1] y Mehlhorn y Sanders [MS08, sec. 5.3] presentan una versión bastante más resumida. Tanto Baase [Baa88, sec. 2.4] como Brassard y Bratley [BB88, sec. 10.1] presentan un material más similar al de Aho et al. También Lee et al. cubren esta cota inferior para el peor caso y caso promedio [LTCT05, sec. 2.4 y 2.6]. Levitin [Lev07, sec. 11.2] explica con bastante detalle la cota de peor caso, así como la cota inferior para la búsqueda en un arreglo ordenado.

Otro tema que no es difícil de encontrar es el de reducciones. Por ejemplo, Brassard y Bratley [BB88, sec. 10.2] presentan muchos ejemplos de problemas con matrices, grafos y aritmética entera y de polinomios. Más llevadera (aunque mucho menor) es la sección que Manber [Man89, sec. 10.4] le dedica a las reducciones, que incluyen una demostración distinta para la cápsula convexa y un par de ejemplos de problemas en matrices. La siguiente sección [Man89, sec. 10.5] es también interesante: habla de los errores típicos al utilizar reducciones para demostrar cotas inferiores. Lee et al. [LTCT05, sec. 2.8] presentan brevemente reducciones, con el ejemplo de la cápsula convexa.

Si tiene interés en el problema mismo de la cápsula convexa, que es bastante famoso, puede ver un buen libro de geometría computacional [dBCvKO08, cap. 11], si bien también se encuentran buenas explicaciones en libros de algoritmos [Man89, sec. 8.4] [Sed92, cap. 25] [Lev07, sec. 4.6] [LTCT05, sec. 4.3].

Knuth [Knu98, pp. 436–453] discute extensamente el tema de la generación de árboles óptimos en tiempo $O(n^2)$, incluyendo los algoritmos de Hu-Tucker (llamados de Garsia-Wachs en esta edición) y el algoritmo cuadrático. Se puede encontrar un tratamiento más pausado de la generación del árbol óptimo en tiempo $O(n^3)$ (sin la mejora a $O(n^2)$) en el libro de Lee et al. [LTCT05, sec. 7.6].

Para códigos de Huffman puede verse un buen libro de Teoría de la Información [CT06, sec. 5.6–5.8], donde también se puede encontrar la desigualdad de Jensen [CT06, sec. 2.6]. Para un tratamiento más algorítmico puede verse, por ejemplo, Sedgewick [Sed92, cap. 22] o Navarro [Nav16, sec. 2.6].

Otras fuentes online de interés:

- jeffe.cs.illinois.edu/teaching/algorithms/notes/28-lowerbounds.pdf
- web.cs.ucdavis.edu/~amenta/w04/dis2.pdf
- www.cs.cmu.edu/afs/cs/academic/class/15210-s12/www/lectures/lecture21.pdf
- algo.kaust.edu.sa/Documents/cs372l04.pdf
- www.cs.princeton.edu/courses/archive/spr08/cos226/lectures/23Reductions-2x2.pdf
- courses.cs.vt.edu/cs5114/spring2010/Bounds.pdf
- www.youtube.com/watch?v=Nz1KZXbghj8

Capítulo 2

Memoria Externa

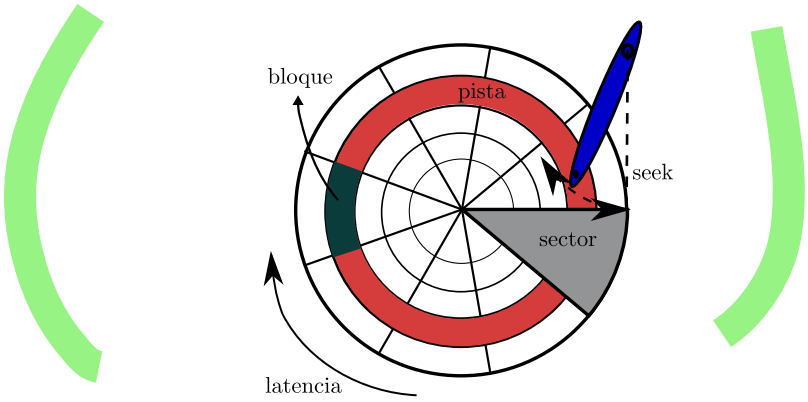
Cuando el volumen de datos a manejar supera la capacidad de la RAM, éstos pueden almacenarse en memoria externa o secundaria (disco, SSD, etc.). En principio, cualquier algoritmo clásico puede usarse sin modificaciones sobre datos en memoria externa. Sin embargo, las operaciones en memoria externa pueden ser hasta un millón de veces más lentas que en la RAM, por lo que vale la pena diseñar algoritmos especialmente adaptados al modelo de costo de estos dispositivos externos. Con un diseño adecuado, los algoritmos en memoria externa pueden ser mucho más rápidos, si bien aún serán considerablemente más lentos que en la RAM.

Estos desarrollos están adquiriendo importancia también para los algoritmos de memoria principal, a medida que las memorias caché se hacen más rápidas en comparación con ella (hoy en día pueden ser hasta 30 veces más rápidas). Un algoritmo para memoria secundaria implementado en RAM suele tener mejor localidad de referencia, y por lo tanto hacer mejor uso del caché que uno clásico, a pesar de que su complejidad no sea mejor.

2.1. Modelo de Memoria Externa

Los discos magnéticos están divididos en pistas (anillos concéntricos) y sectores (limitados por dos radios de círculo consecutivos). La intersección de una pista y un sector es un *bloque* o *página*. Un bloque típicamente almacena unos pocos KBs. Sin embargo, algunos discos tienen varios platos que giran simultáneamente, y la unión del mismo bloque en todos los platos se trata como un único bloque, esta vez de unas decenas de KBs.

El disco magnético escribe a través de un cabezal, que es un dispositivo mecánico que se debe mover a la pista correcta. Esta operación se llama "seek", y toma unas decenas de milisegundos. Luego debe esperar a que el sector pase girando por debajo del cabezal. Este es el "tiempo de latencia", que suma unos pocos milisegundos más. Finalmente, se lee el bloque completo, a una velocidad de unos pocos MBs por segundo. Si se leen bloques contiguos de esa pista, ya no se vuelve a pagar el tiempo de seek ni latencia. Incluso, si se leen bloques



de pistas contiguas, sólo se paga un tiempo mínimo adicional.

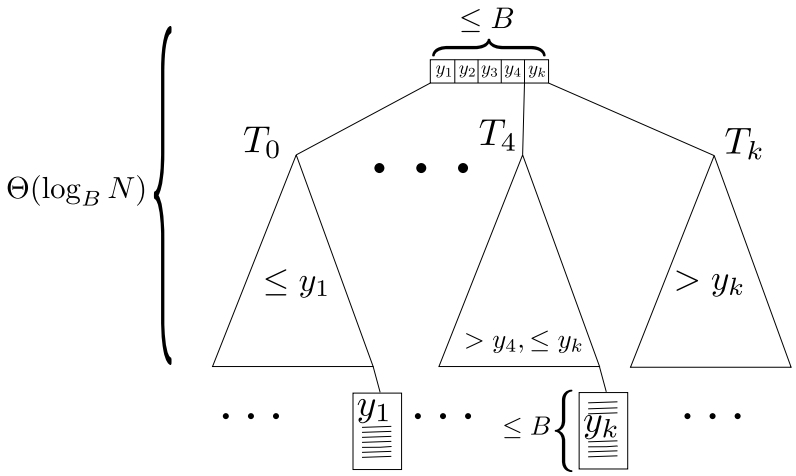
Esto significa que acceder a un elemento en una posición aleatoria cuesta milisegundos, mientras que acceder al elemento al lado de uno leído cuesta microsegundos. Los algoritmos que trabajan en disco son mucho más rápidos si realizan pocos accesos aleatorios y muchos secuenciales. Dado que la RAM accede a los datos en nanosegundos, los accesos aleatorios a disco son un millón de veces más lento que en memoria principal, y los secuenciales son mil veces más lentos. Vel RAM > Vel Disco accesos Seq > Vel Disco accesos aleatorios

En el caso de los SSDs, no existen los componentes mecánicos, por lo que da lo mismo acceder al bloque contiguo que a uno aleatorio. Pero sigue siendo cierto que se leen bloques completos y que su lectura es bastante costosa, unas decenas de microsegundos (diez mil veces más lentos que la RAM).

El modelo de memoria externa que usaremos abstrae de estas dos arquitecturas, las más populares. La memoria externa está formada por bloques de tamaño B. Se leen y escriben bloques completos. La lectura o escritura son tan caras que despreciamos las operaciones del algoritmo en CPU y RAM. Simplemente contamos el número de I/Os, es decir, lecturas y escrituras de bloques. La diferencia entre leer bloques consecutivos o aleatorios no se considera en el modelo. El algoritmo tiene una memoria RAM de tamaño M, que medida en bloques es de tamaño $m = \frac{M}{B}$. El input es de tamaño N, y se presenta en disco en forma contigua, ocupando $n = \frac{N}{B}$ bloques.

Por ejemplo, el costo de un algoritmo que lee secuencialmente un arreglo es $O(n)$. En cambio, uno que lea el arreglo accediendo a sus elementos en un orden aleatorio es $O(N)$, miles de veces mayor en la práctica. En general, lo peor que puede pasar con un algoritmo que se ejecuta en RAM en tiempo $T(N)$ es que al pasarlo a memoria externa también requiera $T(N)$ I/Os, pues éstos son millones de veces más lentos que la operación en RAM. Pero con un diseño adecuado, se puede hacer bastante mejor en muchos casos relevantes.

Veremos estructuras de memoria secundaria para buscar elementos en conjuntos ordenados (árboles de búsqueda) y sin orden (hashing), para colas de prioridad, y algoritmos de ordenamiento.



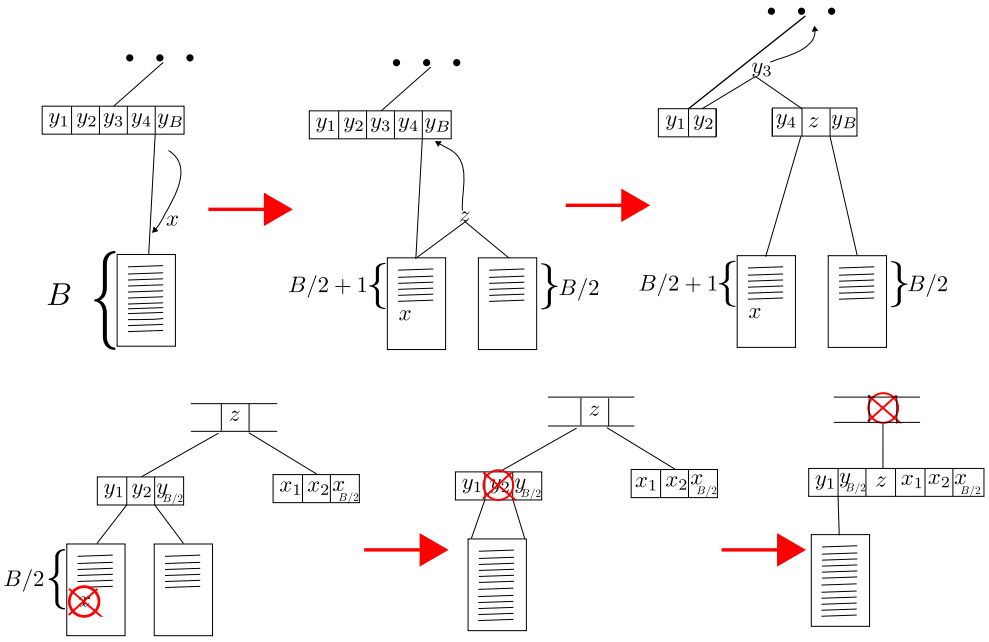
2.2. Árboles B

Los árboles B son una adaptación de los árboles 2–3 a memoria externa, donde cada nodo se almacena en un bloque y entonces se ensancha el nodo a tamaño B para aprovechar que el bloque se lee completo. Los nodos internos del árbol B pueden almacenar hasta $(B-1)/2$ elementos (entendiendo que éstos requieren almacenar $(B-1)/2$ claves y $(B-1)/2+1$ punteros a los nodos hijos). Estas claves están replicadas en las hojas (esta variante, que suele ser la más conveniente, corresponde al llamado árbol B+). Las hojas pueden almacenar hasta B elementos (pues no almacenan punteros), pero suelen almacenar también $B/2$ elementos para poder incorporar un puntero a los datos asociados a cada clave. Incluso pueden almacenar B/c elementos, para alguna constante c, si se elige almacenar los datos asociados directamente en la hoja junto con la clave, para evitar otro acceso aleatorio en disco. Note que los "punteros" son aquí posiciones del archivo en disco donde se almacena el árbol B. Por simplicidad de exposición, diremos que los bloques tienen capacidad de almacenar hasta B claves, tanto en las hojas como en los nodos internos.

Con una capacidad máxima de B claves, el árbol B garantiza que los nodos, salvo la raíz, tienen al menos $B/2$ claves. Cada nodo interno con k claves $y_1, \ldots, y_k$ tiene $k+1$ hijos $T_0, \ldots, T_k$. Todas las claves y del subárbol T_i cumplen $y_i < y \leq y_{i+1}$ (entendiendo $y_0 = -\infty$ e $y_{k+1} = +\infty$). Todas las hojas del árbol B están al mismo nivel, por lo que su altura es $\Theta(\log_B N)$ (está entre $\log_B N$ y $1 + \log_{B/2} N$).

El mecanismo de búsqueda de un elemento x en el árbol B es una extensión clara del mecanismo del árbol 2–3. Se lee el nodo raíz, con sus claves $y_1, \ldots, y_k$, y se busca x entre ellas (en forma binaria o secuencial, no hay diferencia en el modelo de memoria externa). Una vez determinado que $y_i < x \leq y_{i+1}$, la búsqueda continúa en el subárbol T_i. La búsqueda requiere entonces leer $O(\log_B N)$ páginas de disco.

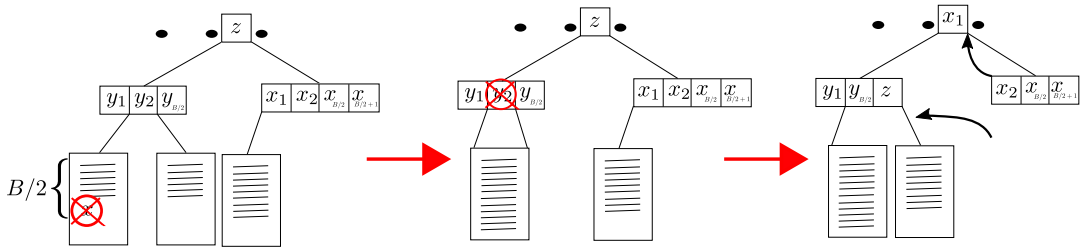
La inserción de un elemento comienza con una búsqueda, donde se identifica la hoja H donde debería estar su clave. El elemento se agrega en la hoja, y si ésta se pasa del tamaño



máximo B, se corta en dos hojas H_1 y H_2 de tamaño $B/2+1$ y $B/2$, respectivamente, y la mediana de las claves (que será la máxima clave almacenada en H_1) se inserta en el nodo padre U, que así reemplaza su antiguo hijo H por dos hijos, H_1 y H_2, separados por la nueva clave. Si el padre U se pasa del tamaño B, es decir pasa a tener $B+1$ claves y $B+2$ hijos, se repite la operación en forma casi idéntica: se corta en dos mitades U_1 y U_2 de $B/2$ claves y $B/2+1$ hijos, y la clave mediana se mueve hacia el padre V de U, reemplazando el nodo U por los nodos U_1 y U_2 que flanquean la nueva clave de V. Si V a su vez se pasa del tamaño máximo, se repite el procedimiento de rebalse de nodos internos. Si finalmente esto ocurre en la raíz, se crea una nueva raíz del árbol con sólo 2 hijos, y la altura del árbol crece en 1.

Para el borrado, se elimina al elemento de la hoja. La clave borrada no necesita eliminarse de los nodos internos, aunque ya no exista más. Si la hoja pasa a tener menos de $B/2$ elementos, entonces se une con su anterior o siguiente hermana, y la clave que las separa en el padre se elimina. Si esto hace que el padre tenga menos de $B/2$ elementos, se repite el proceso de forma similar. La diferencia al unir dos nodos internos es que la clave que los separa en el nodo del padre se baja al nuevo nodo, para separar el último hijo del nodo izquierdo del primero del nodo derecho que se unen. Eventualmente se puede llegar a la raíz, que no requiere tener $B/2$ elementos. Sin embargo, si la raíz queda con cero elementos (y un hijo), se elimina y el hijo pasa a ser la raíz, con lo que el árbol B pierde altura.

En el borrado puede ocurrir que, cuando unimos un nodo con su hermano, el nodo resultante tenga más de B elementos. En ese caso debemos volver a cortar el nodo que hemos creado, por su nueva mediana, y volver a insertar una nueva clave en el padre. En la práctica, esto implica que las claves se redistribuyen entre los nodos hermanos y la clave del padre que los separa se modifica. Cuando esto ocurre, el borrado no necesita seguirse



propagando hacia arriba.

Como puede verse, tanto la inserción como el borrado cuestan $O(\log_B N)$ operaciones de I/O. El árbol B garantiza un porcentaje mínimo de ocupación de las páginas de disco de 50 %. Si los datos se insertan en forma uniformemente aleatoria, el porcentaje promedio de ocupación es de 69 %. Con algunas técnicas más refinadas para evitar cortar hojas cuando rebalsan, la ocupación puede sobrepasar el 80 % promedio.

Por otro lado, note que podríamos mantener los primeros $\Theta(\log_B M)$ niveles del árbol B en memoria principal, de modo que la cantidad de I/Os para búsquedas y modificaciones se reduciría a $O(\log_B \frac{N}{M})$.

2.2.1. Cota inferior

El costo de búsqueda de $O(\log_B \frac{N}{M})$ es óptimo si se busca mediante comparaciones. Demostraremos la cota inferior usando el método del adversario. El modelo es que el algoritmo sabe todo el tiempo el rango del input ordenado en el cual puede estar la clave x que se busca. Si inicialmente partimos con la memoria llena de datos, éstos particionan el input en $M+1$ zonas, y las comparaciones (gratis) con x le permiten al algoritmo establecer que x está en una de las $M+1$ zonas. El adversario se encargará de que se busque una clave que cae en la partición más larga. Esta debe medir al menos $\frac{N}{M+1}$, pues si todas midieran menos, no podrían sumar N. De modo que comenzamos con un rango de ese tamaño.

Cada vez que el algoritmo lee un bloque de B elementos de disco, lee hasta B nuevas claves con las que comparar. El algoritmo y la estructura de datos eligen qué claves son esas. Nuevamente, particionan el rango actual que conoce el algoritmo en $B+1$ subrangos, uno de los cuales pasa a ser el nuevo rango después de las comparaciones. El adversario siempre elige que x esté en el mayor de los rangos, de modo que el rango se reduce en un factor de a lo más $B+1$ por cada lectura. Se deduce que el algoritmo necesita leer al menos $\log_{B+1} \frac{N}{M+1}$ páginas de disco para poder reducir el rango a tamaño 1 y poder responder correctamente.

Con modificaciones simples, el árbol B puede recuperar un rango de elementos, es decir, todos los occ objetos cuyas claves estén en un intervalo $[x_1, x_2]$, en tiempo $O(\log_B \frac{N}{M} + occ/B)$, lo cual es nuevamente óptimo.

2.3. Ordenamiento

Extenderemos el algoritmo de MergeSort a memoria secundaria. MergeSort comienza dividiendo el arreglo en dos y se invoca recursivamente, hasta que los subarreglos que tiene que ordenar son de tamaño 1. Entonces vuelve de la recursión, uniendo los subarreglos ordenados derecho e izquierdo en forma ordenada.

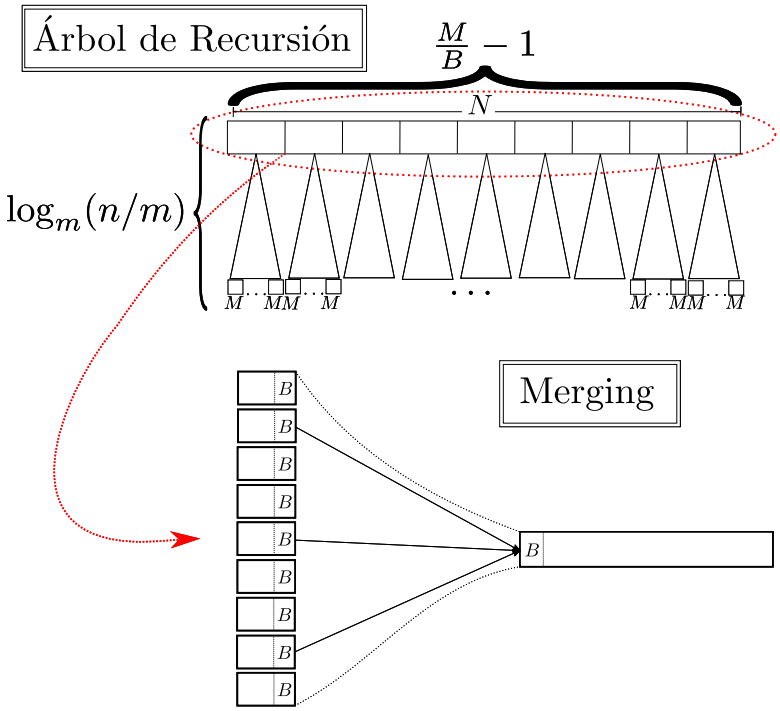
En un entorno de memoria secundaria, tiene sentido detener esta recursión cuando los subarreglos a ordenar son de tamaño B. En este momento se puede leer el bloque de disco, ordenarlo, y reescribirlo a costo $O(1)$ en I/Os. A la vuelta de la recursión, la unión se hace leyendo secuencialmente los dos subarreglos, usando un buffer de tamaño B en memoria para cada subarreglo y otro para el resultado del merging. En total, todas las uniones de un nivel del árbol requieren leer el arreglo completo y reescribirlo, a costo $O(\frac{N}{B}) = O(n)$. Como la recursión se detiene en los subarreglos de tamaño B, la cantidad de niveles en la recursión es $\log_2 \frac{N}{B}$. Es decir, esta variante de MergeSort requiere $O(\frac{N}{B} \log \frac{N}{B}) = O(n \log n)$ I/Os.

Podemos mejorar este costo deteniendo la recursión cuando el subarreglo a ordenar es de tamaño M. En este punto, simplemente se lee el subarreglo a memoria, se ordena (gratis), y se reescribe ordenado. Con la reducción resultante de la cantidad de niveles, el costo de ordenar pasa a ser $O(\frac{N}{B} \log \frac{N}{M}) = O(n \log \frac{n}{m})$ I/Os.

Se consigue una reducción adicional mediante aumentar la aridad del árbol de recursión, es decir, no particionando el subarreglo en dos sino en más. El único límite a la aridad del árbol de la recursión es que, al unir, se necesita tener un buffer de B elementos en memoria por cada archivo que se une, por lo cual éstos no pueden exceder $\frac{M}{B} - 1$. Ahora la cantidad de niveles es $O(\log_{\frac{M}{B}} \frac{N}{M})$, por lo que la cantidad de I/Os del algoritmo se reduce a $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M}) = O(n \log_m \frac{n}{m}) = O(n \log_m n)$ (notar que las últimas dos expresiones difieren sólo en $O(n)$, que es un término de orden inferior).

Esta complejidad es bastante buena en la práctica. Considerando una memoria de GBs y un bloque de KBs, se pueden ordenar PBs (petabytes, 2^{50}) con sólo dos pasadas de lectura y dos de escritura sobre los datos. En la práctica, sin embargo, cuando se usan discos magnéticos, puede ser mala idea llegar realmente a la aridad $\frac{M}{B} - 1$, pues esto aumenta la cantidad de seeks a posiciones aleatorias para leer bloques de muchos archivos distintos. Experimentalmente el óptimo suele ser unir de a unas decenas de archivos por vez. En este caso, ordenar unos PBs puede requerir unas 10 lecturas y escrituras del arreglo completo.

Note que, con la estructura adecuada para la unión, el costo de CPU de este algoritmo es $O(N \log N)$. Cuando se hace la unión de $\frac{M}{B} - 1$ archivos, debe usarse una cola de prioridad en memoria principal para extraer el mínimo entre los primeros elementos de cada uno de los $\frac{M}{B} - 1$ buffers. Eso hace que el costo de unir todos los datos de un nivel sea $O(N \log m)$. Multiplicado por los $\log_m \frac{N}{M}$ niveles, nos da $O(N \log \frac{N}{M})$. A esto debe agregarse el costo de CPU de ordenar los $\frac{N}{M}$ subarreglos de largo M en memoria, en el último nivel de la recursión, $O(N \log M)$, lo que en total nos da $O(N \log N)$.



2.3.1. Cota Inferior

Demostraremos que este algoritmo de ordenamiento es óptimo si se procede por comparaciones. Para ello, volveremos a utilizar la estrategia del adversario. Ya vimos en el capítulo anterior que un algoritmo de ordenamiento debe permitir determinar en cuál de todas las $N!$ permutaciones se ha presentado el input. Nuestro modelo será el conjunto de las permutaciones consistentes con los datos que ha leído el algoritmo hasta ahora. En el estado inicial, este conjunto tiene las $S = N!$ permutaciones posibles, y en los estados finales este conjunto debe tener un único elemento, $S = 1$.

El algoritmo va reduciendo el tamaño S del conjunto de permutaciones factibles a medida que lee un bloque de datos del disco y los compara contra lo que tenga almacenado en memoria. Al leer B valores, si es la primera vez que los ve, el algoritmo puede compararlos entre sí para determinar cuál de los $B!$ posibles ordenamientos entre ellos es el correcto. Asimismo, si guarda otros $M - B$ valores en memoria, puede determinar de cuál de las $\binom{M}{B}$ formas se insertan estos B valores entre los que tiene en memoria. En total, el algoritmo determina la configuración correcta entre las $\binom{M}{B}B!$ que eran posibles antes de leer el bloque (esto es optimista: podría ser que un algoritmo no lograra aprender tanto, pero lo importante es que no puede aprender más que esto).

Cada una de las S permutaciones del input que aún son posibles es compatible con sólo una de estas configuraciones entre las que la lectura del bloque ha permitido distinguir. El conjunto de permutaciones se puede particionar entonces en $\binom{M}{B}B!$ subconjuntos, uno compatible con cada configuración. El adversario puede elegir cuál de estos subconjuntos

es el que resulta compatible con el bloque leído, y tomará el mayor. El mayor subconjunto tiene un tamaño mínimo garantizado de $\frac{S}{\binom{M}{B}B!}$. Es decir, por bien que lo haga el algoritmo, el adversario puede encargarse de que S se reduzca sólo por un factor de $\binom{M}{B}B!$ por cada bloque que lee. Si el algoritmo lee t bloques, entonces a lo sumo puede reducir el tamaño del conjunto inicial a

$$\frac{N!}{\binom{M}{B}^t (B!)^t}.$$

Si seguimos por este camino llegaremos a una cota inferior válida, pero no ajustada. La razón es que hemos sido demasiado optimistas. La cantidad de lecturas de bloques a lo largo del algoritmo debe ser $t \geq n$, pues debe leer todo el input. Y de ellas, sólo las primeras n pueden leer bloques nunca vistos. Por lo tanto, no se puede aprender el orden interno de los elementos del bloque (lo que aporta la componente $B!$) todas las t veces que se lee, sino a lo sumo n veces. (Se pueden escribir bloques nuevos a lo largo del algoritmo, pero como estos bloques se han escrito, entonces estuvieron juntos antes en memoria, por lo tanto el algoritmo ya sabía cómo se ordenaban internamente antes de escribirlos.) En conclusión, si el algoritmo lee t bloques, realmente sólo puede reducir el conjunto de inputs compatibles a

$$\frac{N!}{\binom{M}{B}^t (B!)^n}.$$

Si calculamos ahora cuánto tiene que ser t para que este valor llegue a 1, tendremos

$$t \quad \geq \quad \frac{\log N! - n \log B!}{\log \binom{M}{B}}.$$

Usando la aproximación de Stirling e ignorando términos de orden inferior, tenemos

$$t \quad \geq \quad \frac{N \log N - N \log B}{M \log M - B \log B - (M-B) \log (M-B)},$$

$$t \quad \geq \quad \frac{N \log n}{B \log m + (M-B) \log \frac{M}{M-B}},$$

$$t \quad \geq \quad n \log_m n.$$

Lo que hicimos en el último paso fue usar que $\log \frac{M}{M-B} = \log(1 + \frac{B}{M-B}) = O(\frac{B}{M-B})$ (pues $\ln(1+x) \leq x$), y por lo tanto $(M-B) \log \frac{M}{M-B} = O(B)$, que es de orden inferior al término $B \log m$ que lo acompañaba.

Tenemos entonces que todo algoritmo que ordene en memoria externa por comparaciones requiere $\Omega(n \log_m n)$ I/Os.

2.4. Colas de Prioridad

En problemas de simulación es común tener que manipular cantidades masivas de eventos que no caben en memoria principal (por ejemplo, colisiones entre partículas, donde cada evento dispara otros eventos que deben simularse más adelante). Si estos eventos se pueden manejar con una cola simple, es muy fácil manejarla en disco a un costo de $O(\frac{1}{B})$ por inserción y borrado. Si, en cambio, deben insertarse para ser procesados en un determinado orden, necesitaremos una cola de prioridad en disco. Un argumento simple de reducción nos muestra que manejar una cola de prioridad en disco requiere $\Omega(\frac{1}{B}\log_m n)$ I/Os por operación, pues si no podríamos usarla para ordenar rompiendo la cota inferior que acabamos de demostrar.

2.4.1. Cola de prioridad limitada

Consideremos el siguiente esquema. Usaremos la mitad de la memoria, $\frac{M}{2}$, para mantener una cola de prioridad clásica H. Todas las inserciones ocurrirán en H, gratis. Mientras esta cola no se desborde, no usaremos el disco.

En el momento en que se inserte un nuevo elemento y H esté llena, ésta se ordenará completamente en memoria (gratis) y se almacenará en un archivo en disco, F_1, lo que requerirá $\frac{M}{2B}$ escrituras. Inmediatamente crearemos un buffer de tamaño B en memoria para F_1, donde leeremos su primer bloque. H quedará vacía de nuevo para aceptar nuevas inserciones.

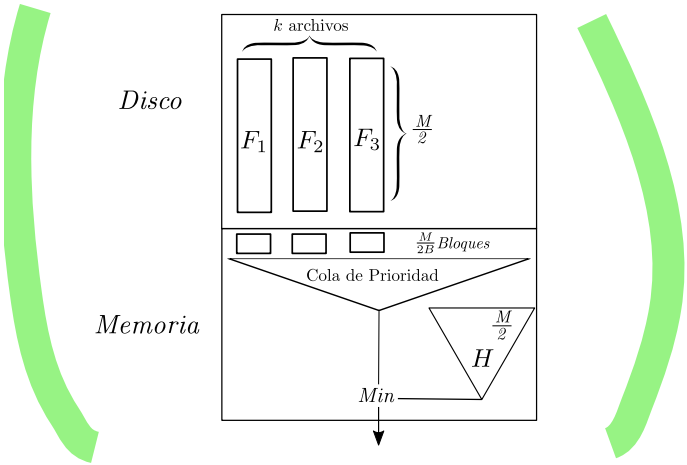
De ahora en adelante, cada vez que extraigamos el mínimo, tendremos que elegir entre el mínimo de H y el primer elemento del buffer de F_1. Una vez que leímos todo el buffer de F_1, lo volvemos a llenar leyendo el siguiente bloque de B elementos.

Como H sigue recibiendo inserciones, puede volverse a llenar. En este caso lo ordenamos nuevamente y lo escribimos en un nuevo archivo, F_2. En general, tendremos k archivos ordenados $F_1, \ldots, F_k$, y las extracciones de mínimo tendrán que considerar el mínimo entre el mínimo de H y los mínimos de cada F_i. Esto se hace fácilmente en tiempo de CPU $O(\log k)$ con una pequeña cola de prioridad que mantiene los primeros elementos de cada F_i y los reemplaza por el siguiente de su buffer cuando éstos son extraídos.

Note que en todo momento los archivos F_i pueden estar a medio leer. Podríamos pensar en un mecanismo más sofisticado que eliminara los archivos leídos, o los uniera cuando se hicieran pequeños, pero aquí mantendremos la simplicidad: los archivos F_i se crean y se van leyendo, y nunca se eliminan o unen.

Considerando que tenemos $\frac{M}{2}$ espacio de memoria para los buffers, tenemos un límite de $k \leq \frac{M}{2B}$. Esto significa que tenemos un límite de $N \leq k \cdot \frac{M}{2} + \frac{M}{2} \leq \frac{M}{2}(\frac{M}{2B} + 1) = O(\frac{M^2}{B})$ al total de elementos que pueden ser insertados en esta estructura (en el peor caso; en la práctica muchos podrían eliminarse antes de pasar a un archivo F_i). Con una memoria de GBs y un B de KBs, esto equivale a PBs (petabytes).

Para analizar el costo de las operaciones, consideremos lo que nos puede costar un elemento desde que es insertado hasta que es extraído. La inserción es gratis, pero el elemento puede finalmente ser enviado a un archivo F_i, donde es escrito junto con otros $B-1$ elemen-



tos, por lo que podemos cobrarle $\frac{1}{B}$ escrituras. Luego, puede ser leído de este archivo a su buffer en memoria, junto con otros B elementos, por lo que podemos cobrarle $\frac{1}{B}$ lecturas. En total, cada operación cuesta $O(\frac{1}{B})$ I/Os. Esto, por supuesto, es en un sentido *amortizado*: muchas operaciones son gratis, y de repente una inserción provoca un costo de $O(\frac{M}{B})$ para escribir un archivo F_i completo. En un esquema más sofisticado, podemos "deamortizar" el costo mediante escribir este archivo poco a poco, dividiendo H en dos colas de tamaño $M/4$, de manera que cuando una se llena empezamos a usar la otra y vamos escribiendo la que se llenó poco a poco a disco, a lo largo de las sucesivas inserciones que siguen. Debemos asegurar que, para cuando la segunda cola se llene, la primera ya se habrá vaciado y puedan intercambiar sus roles.

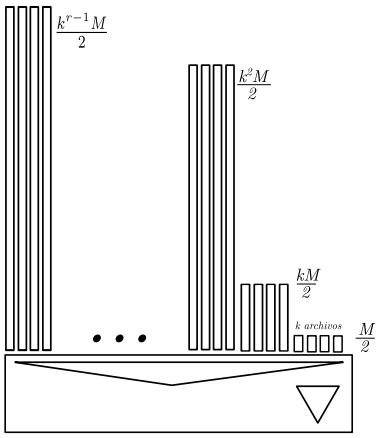
Aún en sentido amortizado, esta complejidad parece violar la cota inferior: podríamos ordenar en disco en tiempo $O(\frac{N}{B})$ mediante insertar los N elementos y luego extraerlos de esta cola de prioridad. Esto es efectivamente cierto, pero dentro de la limitación de $N = O(\frac{M^2}{B})$. Bajo este supuesto, la complejidad $\Theta(n \log_m \frac{n}{m})$ de ordenar es efectivamente $\Theta(n)$.

2.4.2. Cola de prioridad general

En caso de que debamos manejar más elementos que los permitidos por el esquema anterior (por ejemplo, no siempre la estructura tendrá permitido usar toda la RAM, con lo cual la limitación podría ser más notoria), extenderemos el esquema previo mediante una secuencia creciente de *grupos* de archivos.

Tendremos un número máximo k de archivos, como antes, pero éstos serán los archivos $F_1^1, \ldots, F_k^1$ del primer grupo. Cuando se intente crear el archivo F_{k+1}^1, lo que haremos será unir todos los archivos del grupo actual en uno nuevo, F_1^2, de tamaño máximo $k \cdot \frac{M}{2}$. Con ello quedan libres todos los archivos F_i^1 y se pueden volver a llenar. Una vez que se creen todos los archivos $F_1^2, \ldots, F_k^2$ y se necesite crear uno nuevo, se unirán todos en un nuevo archivo, F_1^3, de tamaño $k^2 \cdot \frac{M}{2}$, y se vaciarán todos los F_i^2. Y así sucesivamente.

Cada unión de archivos $F_1^i, \ldots, F_k^i$ para construir un F_j^{i+1} cuesta $O(|F_j^{i+1}|/B) = O(k^i m)$,



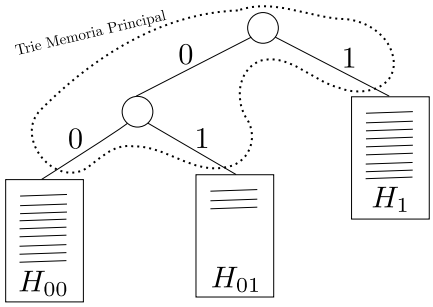
es decir, $O(\frac{1}{B})$ por elemento unido. Si en total construimos r grupos, el costo amortizado de una operación es $O(\frac{r}{B})$, pues a lo largo de su vida en la estructura, un elemento insertado puede ser escrito, luego unido $r-1$ veces, y finalmente leído.

Para poder crear el primer elemento del grupo r debemos haber insertado $N = \sum_{i=0}^{r-1} k^i \cdot \frac{M}{2} = \Theta(k^r M)$ elementos, por lo tanto el número de grupos que podemos llegar a producir es $r = \log_k \frac{N}{M} + O(1)$. Debemos tener $k \cdot r$ buffers en memoria para poder ir extrayendo los mínimos de cada archivo de cada grupo, por lo cual necesitamos que $\frac{M}{2} \geq krB$, es decir, estamos limitados a $2kr \leq m$. Para tener tiempo óptimo necesitamos que $r = O(\log_m \frac{n}{m})$, es decir, $\log k = \Theta(\log m)$. Podemos entonces elegir $k = \Theta(m^\alpha)$ para algún $0 < \alpha < 1$ constante (el costo de las operaciones se multiplicará por $\frac{1}{\alpha}$). Dada la restricción $2kr \leq m$, esto significa que debe cumplirse $m^\alpha \log_m n = O(m)$, es decir $\log n = O(m^{1-\alpha} \log m)$.

Esta condición es bastante generosa en la práctica. Por ejemplo, considérese sólo 1 MB de memoria y 1 KB de tamaño de bloque, con lo cual $m = 2^{10}$, y $\alpha = 1/2$. Para manejar 1 YBs (un yottabyte, $N = 2^{80}$) de datos, usemos $k = m^\alpha = 2^5$ y obtenemos $r = \log_k \frac{N}{M} = 12$, con $2kr = 768 < 1024 = m$, y el esquema nos cabe en memoria a sólo el doble del costo óptimo (que sería $\log_m \frac{n}{m} = 6$).

2.5. Hashing

Además de buscar todos los objetos en un rango, el árbol B puede encontrar el predecesor o sucesor de un elemento en el conjunto de las claves, mediante una modificación simple del algoritmo de búsqueda. Cuando estas capacidades no son necesarias y sólo se desea poder encontrar un elemento insertado, podemos diseñar estructuras basadas en hashing que, bajo ciertos supuestos razonables, permitan buscar haciendo $O(1)$ accesos al disco en promedio. Para esto bastaría con implementar una tabla normal de hashing en disco. Sin embargo, una tabla normal de hashing requiere o bien conocer de antemano la cantidad de elementos que se almacenarán, para dar un tamaño adecuado a la tabla, o bien incrementar periódicamente el tamaño de la tabla. Esto tiene un costo importante y es especialmente indeseable en



memoria externa, que es mucho más lenta. Asimismo, en memoria externa, donde los datos son mucho más masivos y posiblemente persistentes, es menos probable que se tenga una idea aproximada del tamaño de datos que se deberán manejar.

Presentaremos dos esquemas de hashing que buscan ofrecer costo de operación $O(\lceil t/B \rceil)$ basándose en una función de hashing $h(\cdot)$ clásica que produzca $O(t)$ colisiones ($t = O(1)$ si la función está bien diseñada).

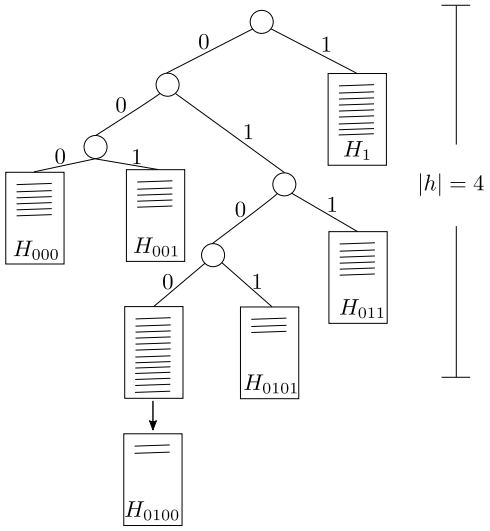
2.5.1. Hashing Extendible

Este esquema de hashing funciona, en promedio, cuando se almacenan $N = O(MB)$ datos en total, es decir, unas miles de veces el tamaño de la memoria principal. Inicialmente, la estructura es una única página en disco, H, donde los datos se insertan de cualquier manera, al costo de $O(1)$ I/Os (o incluso gratis si la tenemos en memoria principal). Las búsquedas se hacen leyendo la página H y buscando secuencialmente la clave que se desea.

Una vez que esta página se llena, la siguiente inserción provoca que la dividamos en dos, H_0 y H_1. Para ello, releemos cada dato y de H y calculamos $h(y)$. Según el primer bit de $h(y)$ sea 0 ó 1, insertamos el elemento en H_0 o H_1, respectivamente. Luego, creamos un nodo en memoria principal con dos hijos: el izquierdo apunta a la página de disco donde almacenamos H_0, y el derecho a la de H_1. Supongamos que, más adelante, la página de H_0 rebalsa por una inserción. Recorreremos todos los elementos y de H_0 y consideraremos el segundo bit de $h(y)$ para separar los elementos en dos hojas, H_{00} y H_{01}. La hoja de H_0 se reemplazará entonces por un nodo interno, cuyos hijos izquierdo y derecho serán, respectivamente, H_{00} y H_{01}.

En general, tendremos un árbol en memoria de tipo *trie* (que volveremos a ver en el capítulo de universos discretos), con $k-1$ nodos y k hojas, donde cada hoja almacena datos en una página de disco. Las búsquedas por una clave x parten por calcular $h(x)$, y usan sus bits para recorrer el trie, desde la raíz hasta llegar a una hoja. En ese momento leen la página correspondiente del disco y buscan x secuencialmente entre las claves. El costo de búsqueda es, en principio, siempre de 1 lectura, y el de inserción agrega 1 ó 2 escrituras.

Note que, cuando se divide una página, no hay garantía de que la división sea equitativa. Una hoja podría quedar con más elementos que la otra. Esto significa que no hay una ocupación mínima garantizada, y que en particular no podemos garantizar que la cantidad

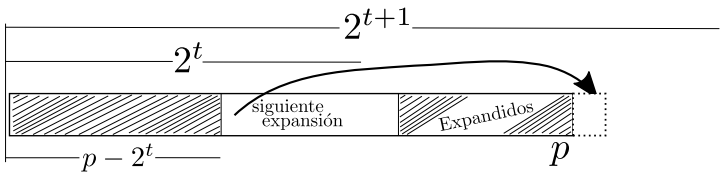


de nodos del trie es $O(k) = O(N/B)$. Es decir, con mala suerte se nos podría acabar la memoria disponible para un N mucho menor que $\Theta(MB)$. En promedio, sin embargo, si consideramos que los valores $h(y)$ son aleatorios, las páginas estarán llenas a un 69 %.

Incluso podría ocurrir que, al dividir una página, todas las claves se fueran a una de las dos páginas. En ese caso, no se necesita crear una página vacía en disco. Basta que el puntero desde el árbol sea nulo para indicar esa situación, y deberemos volver a particionar la otra página, que continuará rebalsada, todas las veces que sea necesario. Note que, aún en este caso, realizamos solamente 2 escrituras a disco.

También puede ocurrir que la página que rebalsa ya sea de profundidad $|h(y)|$, es decir, que ya se hayan usado todos los bits de la función de hashing. Esto equivale a decir que tenemos más de una página de elementos que *colisionan* en el hash provisto por $h(\cdot)$. Si bien esto no debería ocurrir con una $h(\cdot)$ bien diseñada, debe haber una provisión para este caso. Lo que se hace es tener una lista enlazada de las páginas que rebalsan en el último nivel. Por ello, si la función $h(\cdot)$ produce una colisión entre t elementos, que requerirían tiempo $O(t)$ para buscarse en memoria principal, el costo en disco será $O(\lceil t/B \rceil)$ lecturas.

Finalmente, para borrar un elemento, debe encontrarse su página y eliminarlo. Luego de esto, puede ocurrir que la página quede vacía, en cuyo caso se elimina y el puntero en el árbol se hace nulo. Más probable, sin embargo, es el caso en que el nodo hermano en el árbol también sea una hoja y que ambas quepan en una sola. En este caso se pueden unir y reemplazar el nodo padre de ambas páginas por la página unida. Esta unión puede hacerse exigiendo que la nueva página esté, por ejemplo, $\frac{2}{3}$ llena, para evitar secuencias de uniones y divisiones muy seguidas para una página que almacena cerca de B elementos y sufre inserciones y borrados consecutivos. Asismismo, debe verificarse que la nueva página no tenga como hermano un puntero nulo, ya que en ese caso se puede reemplazar al nodo padre por la hoja creada. Esto puede ocurrir repetidamente para varios ancestros de la página creada. En total, sin embargo, un borrado requiere de 1 ó 2 lecturas y 1 escritura.



2.5.2. Hashing Lineal

El hashing lineal provee algunas ventajas sobre el extendible. Para comenzar, requiere almacenar sólo $O(1)$ datos en memoria, por lo que puede manejar conjuntos arbitrariamente grandes de datos. Segundo, permite controlar el porcentaje de llenado de los bloques, o bien el costo promedio de búsqueda (pero no ambos).

Pensemos primero que el archivo de hashing en disco tuviera siempre 2^t páginas. Un elemento y está guardado en la página número $h(y)$ mód 2^t (es decir, los t bits más bajos de $h(y)$). Si algunas páginas rebalsan durante las inserciones, les creamos una lista enlazada de páginas de rebalse. Si, luego de un rebalse, notamos que el costo de búsqueda (es decir, 1 más el largo promedio de las listas de rebalse) se ha hecho demasiado alto, *expandimos* la tabla. Expandir significa duplicar su tamaño a 2^{t+1}. Cada página i, con $0 \leq i < 2^t$, se recorre y sus elementos y se reinsertan en la página $h(y)$ mód 2^{t+1}. Esto significa que una parte de los elementos se quedan en la página i, mientras que otros se insertan en la página $i + 2^t$.

El hashing lineal funciona de esa forma, pero realiza el proceso de expansión de manera gradual. En general, el archivo contiene p páginas, con $2^t \leq p < 2^{t+1}$. Inicialmente tenemos $p = 1$ páginas y $t = 0$. Las páginas $0 \leq i < p - 2^t$ ya fueron expandidas, y repartidas entre las páginas i e $i + 2^t$, mientras que las páginas $p - 2^t \leq i < 2^t$ aún no han sido expandidas.

Para buscar un elemento y en el caso general, se calcula $k \leftarrow h(y)$ mód 2^{t+1}. Si $k < p$, entonces se lee la página k (y su posible lista de rebalse) para buscar la clave y en ella, secuencialmente. Si $k \geq p$, sin embargo, la página k aún no ha sido creada por el proceso de expansión, por lo cual el proceso de lectura debe realizarse en cambio en la página $k \leftarrow k - 2^t$.

Cuando se cumple una determinada condición (por ejemplo, el costo promedio de búsqueda supera un cierto valor permitido), expandimos *la siguiente página*, es decir, la página $p - 2^t$ (¡que no es necesariamente la que produjo el rebalse que llevó a exceder el costo promedio de búsqueda permitido!). Esta página se lee y sus elementos y se reinsertan en las páginas $h(y)$ mód 2^{t+1}, es decir, se reparten entre la misma $p - 2^t$ y la nueva página p, que se agrega al final del archivo. Al terminar la expansión, hacemos $p \leftarrow p + 1$, y si resulta que $p = 2^{t+1}$, entonces hemos completado una expansión y nos preparamos para la siguiente: $t \leftarrow t + 1$. Para eliminar un valor, éste se busca en la tabla y simplemente se elimina. En caso de estar en una lista de rebalse, puede usarse su espacio para mover un elemento desde la última página de la lista, de modo de poder liberar apenas se pueda esta última página y reducir así el tiempo promedio de búsqueda. Si se eliminan suficientes elementos, puede resultar que la tabla pueda *contraerse* sin que se exceda el costo promedio de búsqueda máximo permitido. Para realizar una contracción, primero se hace $t \leftarrow t - 1$ si $p = 2^t$, y luego se hace $p \leftarrow p - 1$.

Luego, se agregan todos los elementos de la página p a la página $p - 2^t$, procediendo a eliminar la página p. Note que esto puede hacer rebalsar la página $p - 2^t$, o alargar su lista de rebalse.

Note que una expansión o una contracción no necesariamente cambiarán la condición que las disparó acerca del costo promedio de búsqueda. En general es preferible, para evitar que una inserción o borrado disparen muchas expansiones o contracciones, realizar de todas maneras una sola, y dejar que la siguiente operación dispare nuevamente una expansión o contracción, hasta que la situación se resuelva.

Se pueden usar otros criterios en vez del máximo costo promedio de búsqueda. Por ejemplo, puede permitirse un mínimo porcentaje de llenado de las páginas, de modo de contraer cuando éste se hace demasiado bajo (y expandir cuando es posible sin violar el criterio). Este criterio se contrapone al de mantener un costo de búsqueda máximo, por lo que sólo se puede controlar uno de los dos (o puede usarse una combinación de criterios). En general el hashing lineal se comporta mejor que el extendible, aunque no suele garantizar un solo acceso por lectura.

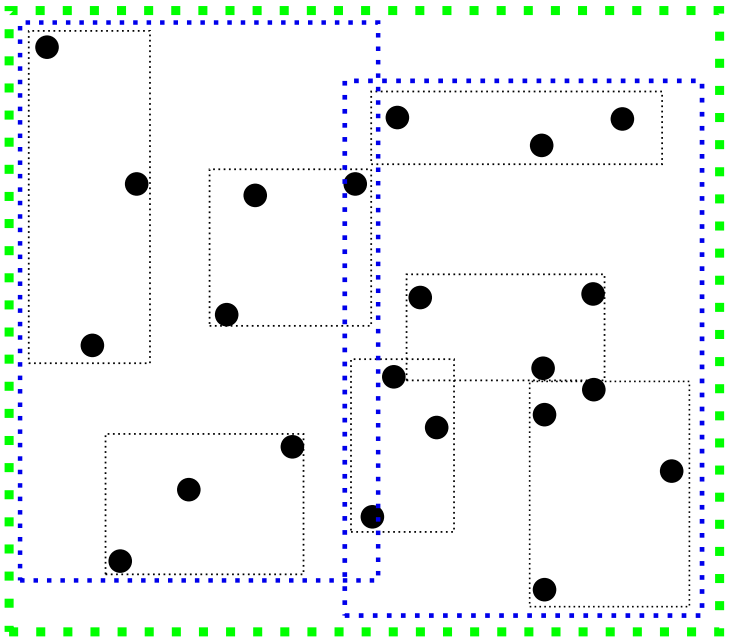
2.6. R-trees

El R-tree es la estructura de datos más popular para almacenar puntos o hiperrectángulos, en 2 o más dimensiones. Es una extensión de la idea del B-tree, en el sentido de que los nodos usan bloques de disco garantizando una fracción mínima de llenado, tiene altura $O(\log_B N)$, las hojas están todas al mismo nivel, y usa mecanismos similares de inserción y borrado.

Las "claves" son *minimum bounding boxes (MBBs)*, es decir, el menor hiperrectángulo que encierra un conjunto de objetos. El R-tree puede representar objetos complejos, y la clave de búsqueda es su MBB (como caso particular, podemos también almacenar puntos). Cada nodo interno tiene k claves y k hijos. La clave y_i que almacena para su hijo T_i es el MBB de los MBBs almacenados en la raíz de T_i. Dado un parámetro $0 < \alpha \leq \frac{1}{2}$, el R-tree garantiza que todo nodo u hoja, excepto la raíz, tiene entre αB y B claves.

El R-tree permite encontrar todos los objetos cuyos MBBs se intersecten con un hiperrectángulo de consulta. La forma de proceder es leer la raíz, comparar la consulta con los k MBBs que almacena, y recursivamente continuar la búsqueda en todos los subárboles T_i tal que y_i se intersecta con la consulta. Cuando se llega a las hojas, los MBBs intersectados se reportan. Asimismo, se puede usar para encontrar todos los objetos del R-tree que contienen al de la consulta, mediante entrar en todos los hijos cuyos MBBs contengan a la consulta.

Note que la consulta puede entrar a cero o más hijos de un nodo, por lo que el tiempo de búsqueda no es $O(\log_B N)$. Podemos obtener una complejidad promedio si consideramos una probabilidad fija p de que la consulta se intersecte con un MBB. Entonces el tiempo promedio cumple la recurrencia $T(N) = 1 + pB \cdot T(N/B)$, cuya solución es $O(N^{1-\log_B(1/p)})$. Es decir, la complejidad es de la forma $O(n^\beta)$ para un $0 < \beta < 1$ que depende de la probabilidad de intersección. Es por ello que es importante lograr que los MBBs sean lo más pequeños



posible, mediante una adecuada política de inserción y borrado de objetos.

Para insertar un objeto x, partimos de la raíz y vemos si está completamente contenido en algún MBB. Si lo está, elegimos el de menor área y continuamos. Si no, calculamos en qué hijo T_i tendríamos que incrementar menos el área de la clave y_i al convertirla en $y_i' = MBB(y_i \cup x)$, reemplazamos y_i por y_i' y continuamos la inserción en T_i. Finalmente, al llegar a una hoja, agregamos x a los objetos.

En caso de que la hoja pase a tener $B + 1$ objetos, debemos partirla en dos hojas de modo de minimizar la suma de las áreas de ambos MBBs y que ninguna tenga menos de αB objetos. Note que en el modelo de memoria externa podríamos considerar las $\Theta(2^B)$ particiones posibles y seleccionar la mejor, lo que sería "gratis" en términos de I/O, pero esto es impracticable en la realidad por su costo de CPU. En cambio, se utilizan heurísticas. Una clásica, llamada "quadratic split", hace lo siguiente:

- Escoge dos claves y e y' lo más alejadas posible, es decir, que maximicen las áreas de $MBB(y \cup y') - MBB(y) - MBB(y')$. Esto toma tiempo $O(B^2)$ en memoria principal. Estas claves serán los primeros elementos de las dos nuevas hojas.

- Va insertando las demás claves, eligiendo en cada paso la que incremente menos el área al insertarla en el MBB de la hoja de y o en el de la hoja de y'. En caso de empate, puede escoger la hoja de menor área o la de menos elementos. Esto también cuesta $O(B^2)$ operaciones en memoria principal.

- Cuando una hoja llegue a $(1 - \alpha)B$ elementos, los demás van a la otra.

Este mecanismo se usa también cuando los nodos internos rebalsan. En total, una inserción cuesta $O(\log_B N)$ I/Os, y $O(B^2 \log_B N)$ tiempo de CPU.

Para borrar un elemento, se elimina de su hoja y se calcula su nuevo MBB, que puede decrecer. A la vuelta de la recursión, se pueden ir recalculando los MBBs de los ancestros del nodo, al costo de una nueva escritura del bloque. Cuando una hoja tiene menos de αB elementos, en vez de intentar algo parecido al B-tree, es decir, unirla con una hoja vecina y de ser necesario volverlas a partir, lo que hace el R-tree es eliminarla completamente y reinsertar todos los elementos en el árbol. Esto suele mejorar el empaquetamiento de objetos en MBBs. Note que esto puede ocurrir también con subárboles completos, cuando un nodo interno queda con menos de αB claves.

Almacenando los primeros $\Theta(\log_B M)$ niveles en memoria principal, el costo de inserción se reduce a $O(\log_B \frac{n}{m})$ I/Os.

2.7. Ficha Resumen

- Árbol B: $O(\log_B \frac{n}{m})$ para insertar, borrar y buscar. Óptimo para buscar si se procede por comparaciones. Ocupación promedio 69 %. Permite recuperar todos los *occ* objetos en un rango en tiempo óptimo $O(\log_B \frac{n}{m} + occ)$, y encontrar el predecesor y sucesor de un elemento en tiempo $O(\log_B \frac{n}{m})$.

- Ordenamiento: $O(n \log_m n)$ (o, equivalentemente, $O(n \log_m \frac{n}{m})$), lo que es óptimo si se procede por comparaciones.

- Cola de prioridad: $O(\frac{1}{B} \log_m n)$ (amortizado) si $\log \frac{n}{m} \leq m^\alpha$ para una constante $0 < \alpha < 1$, lo que es óptimo si se procede por comparaciones.

- Hashing extendible: para $N = O(MB)$. Inserta, borra y busca en $O(1)$ promedio, con un buen hash. Con uno que produce t colisiones, el costo es $O(\lceil t/B \rceil)$. Ocupación promedio 69 %.

- Hashing lineal: para cualquier N. Se puede acotar el costo promedio o la ocupación promedio a costa de la otra medida.

- R-tree: $O(\log_B \frac{n}{m})$ para insertar y (salvo cuando se debe reinsertar un nodo) para borrar. Para encontrar todos los que intersectan o contienen un rectángulo de consulta, $O(n^\beta)$ en promedio, para una constante β que depende de la consulta y de los datos.

2.8. Material Suplementario

Probablemente la descripción del modelo de costo en memoria externa más recomendable es la de Vitter [Vit08, cap. 2] o la de Meyer et al. [MSS03, cap. 1], por su nivel de detalle

y completitud. Incluyen también otros modelos interesantes, que incluyen paralelismo y la jerarquía completa de memoria. Cormen et al. [CLRS01, cap. 18] describen el modelo de costo con mucho menos detalle, pero aún razonablemente bien. Casi todos los otros libros mencionados en esta sección describen también el modelo de costo, aunque generalmente con aún menos detalle.

Cormen et al. [CLRS01, cap. 18] explican detalladamente los árboles B. También los describen Weiss [Wei95, sec. 4.7] y Sedgewick [Sed92, cap. 18], aunque con bastante menos detalle. Aho et al. [AHU83, cap. 11] también describen los árboles B, precedidos de una discusión que es interesante para convencerse de la necesidad de este tipo de estructuras cuando la masividad de los datos hace que las ideas más simples fracasen. Vitter [Vit08, cap. 11] también describe los árboles B con bastante detalle, así como otras variantes que son útiles, por ejemplo, para realizar muchas modificaciones en grupo. Meyer et al. [MSS03, sec. 2.3] también presentan los árboles B en detalle, e incluyen varias variantes de interés. Asimismo, explican brevemente la cota inferior para buscar en disco [MSS03, sec. 1.5.2].

Weiss [Wei95, sec. 7.11] describe el MergeSort para memoria externa, pero se centra en un modelo de k cintas en vez de discos. En las cintas, el acceso sólo puede ser secuencial. Finalmente el algoritmo no es muy distinto al que vimos, si bien hay más detalles distractivos de lo conveniente. También usando cintas, Baase [Baa88, sec. 2.8] y Sedgewick [Sed92, cap. 13] describen MergeSort con bastante detalle y discuten varios aspectos prácticos. Una descripción más moderna y simple se puede encontrar en Mehlhorn y Sanders [MS08, sec. 5.7], donde además se presenta SampleSort, una variante muy sencilla que también funciona bien en la práctica. Meyer et al. [MSS03, sec. 3.2.2] discuten con bastante detalle variantes de MergeSort y de SampleSort. Vitter [Vit08, cap. 5 y 6] dedica un largo capítulo a técnicas avanzadas de ordenamiento, si bien el material no es tan aconsejable para leer sobre MergeSort básico. En el capítulo 6, Vitter explica la cota inferior para ordenar en disco. Meyer et al. [MSS03, sec. 1.5.1] cubren en detalle esta cota inferior, junto con la del problema relacionado de permutar un arreglo en disco.

Meyer et al. [MSS03, sec. 2.1] cubren estructuras elementales en disco, como pilas, colas y listas, así como una cola de prioridad basada en una variante de los árboles B (los *buffer trees*) que obtiene resultados similares a los vistos en el capítulo, pero que son siempre óptimos [MSS03, sec. 2.3.6]. La versión que presentamos se describe, en su variante simple, en Mehlhorn y Sanders [MS08, sec. 6.3], donde dan referencias a la versión completa.

Aho et al. [AHU83, sec. 11.3] describen un hashing fijo en disco, sin un mecanismo eficiente para crecer cuando las listas de rebalse se hacen demasiado largas. Weiss [Wei95, sec. 5.6] describe el hashing extendible, si bien, como en casi toda la literatura, usa en memoria una tabla que se duplica para abarcar la altura máxima en vez de una estructura de trie. Lo mismo hace Sedgewick [Sed92, cap. 18], con bastante detalle. Vitter [Vit08, sec. 10.1] también describe el hashing extendible con cierto detalle, y luego describe muy brevemente el hashing lineal [Vit08, sec. 10.2]. Samet [Sam06, ap. B] describe el hashing extendible usando tries, así como el hashing lineal y otras variantes, con mayor detalle. Meyer et al. [MSS03, sec. 2.4] describen el hashing lineal y extendible, así como varias otras variantes de hashings capaces

y no capaces de hacer crecer las tablas.

Existen muchos otros algoritmos para memoria secundaria. Por ejemplo, Mehlhorn y Sanders [MS08, sec. 11.5] describen algoritmos para árboles cobertores mínimos en memoria externa. Dos excelentes referencias para más algoritmos y estructuras de datos en memoria externa son los libros de Vitter [Vit08] y de Meyer et al. [MSS03], que incluyen temas de matrices, geometría, grafos, textos, y técnicas generales de diseño.

Para los R-trees es mejor consultar un libro de estructuras de datos espaciales [Sam06, sec. 2.1.5.2.3–2.1.5.2.7].

Otras fuentes online de interés:

- El libro de Vitter [Vit08], `www.ittc.ku.edu/~jsv/Papers/Vit.IO_book.pdf`
- El libro de Meyer et al. [MSS03], `link.springer.com/content/pdf/10.1007/3-540-36574-5.pdf`
- `algo2.iti.kit.edu/download/mem_hierarchy_02.pdf` a `algo2.iti.kit.edu/download/mem_hierarchy_06.pdf`.
- `www.daimi.au.dk/ large/ioS09/`
- `people.mpi-inf.mpg.de/~mehlhorn/AlgorithmEngineering/ExternalMemorySlides.pdf`
- `164.100.133.129:81/eCONTENT/Uploads/8.0 Data Structures and Algorithms for External Storage.pdf`
- `ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-851-advanced-data-structures-spring-2012/calendar-and-notes/MIT6_851S12_L7.pdf`
- `web.stanford.edu/class/cs145/cs145-notebooks-2016/lecture-11-12/Lecture_11-12_Indexes.pdf`
- `www.imada.sdu.dk/~rolf/Edu/DM808/F08/`
- `www.youtube.com/watch?v=py4z_v9dfzQ` y `www.youtube.com/watch?v=KZua1GbIGr8`

Capítulo 3

Análisis Amortizado

El análisis amortizado es una técnica que permite analizar el costo de una *secuencia* de n operaciones, cuando éste es menor de lo que se obtiene tomando el peor caso de una operación y multiplicándolo por n. En este caso, diremos que el *costo amortizado* de cada operación es el costo total de la secuencia de operaciones dividido n. Note que esto no es lo mismo que análisis de caso promedio: se considera la peor secuencia posible de n operaciones (aunque, independientemente, podríamos hablar de costo promedio amortizado).

Veremos tres técnicas de análisis amortizado. Según el caso, puede ser más natural utilizar una que otra. Las ejemplificaremos con algunos casos sencillos. Después veremos algunos algoritmos y estructuras de datos relevantes donde el análisis amortizado es fundamental para comprender sus costos.

3.1. Técnicas

Comencemos con un problema de juguete para ejemplificar las técnicas a medida que las describimos. Supongamos que tenemos una pila donde permitimos las operaciones $push(x)$ (que apila x) y $multipop(k)$ (que desapila k elementos con algún propósito, por ejemplo entregar su suma). La operación $push(x)$ cuesta $\Theta(1)$ y la operación $multipop(k)$ cuesta $\Theta(k)$ (supondremos que cuando se ejecuta es porque hay al menos k elementos en la pila). La pregunta es ¿cuánto puede costar una secuencia de n operaciones en una pila vacía?

Un análisis de peor caso nos muestra que, luego de haber realizado $\Theta(n)$ *push*'s, un *multipop* nos puede costar $\Theta(n)$. Por lo tanto, una secuencia de n operaciones partiendo de una pila vacía puede costar $O(n^2)$. Si bien esto es formalmente cierto, la cota está lejos de ser ajustada. Usaremos tres tipos de argumentos para mostrar que en realidad n operaciones sólo pueden costar $O(n)$, es decir, el costo amortizado por operación es $O(1)$ (si bien es verdad que algún *multipop* puede costar $\Theta(n)$).

Análisis global. La primera forma de verificar esto es notar que, en n operaciones, sólo se pueden haber apilado n elementos mediante *push*'s. Por ello, si bien algún *multipop* puede costar casi n, la suma de todos los *multipop*'s sólo puede costar n, pues todo elemento desapilado se debe haber apilado alguna vez. Es decir, con n operaciones a lo sumo podemos haber apilado n elementos y luego haberlos desapilado. Si tomamos el costo de *push* como 1 y de *multipop*(k) como k, el costo total de las n operaciones es a lo más $2n = O(n)$.

Esta técnica se llama *análisis global*: observamos los costos de toda la secuencia no paso a paso sino globalmente, para deducir alguna propiedad que permita acotar el costo total. Ésta es la más sencilla de las técnicas, aunque no siempre es fácil encontrar una visión global que haga obvio el costo total.

Contabilidad de costos. La segunda forma es notar que cada elemento que se saca con *multipop* debe haber entrado en la pila con un *push* alguna vez. Podemos entonces cobrarle 2 operaciones al *push*, y cobrarle cero al *multipop*. Visto de otro modo, le estamos cobrando por adelantado al elemento x de *push*(x) el costo que producirá cuando más adelante participe de un *multipop*(k). Queda entonces claro que una secuencia de n operaciones no puede costar más que $2n$, pues no puede haber más de n operaciones de *push*.

Esta técnica se llama *contabilidad de costos*: repartimos el costo real de alguna forma, entre operaciones, objetos, etc. para que resulte más fácil de sumar. Podemos, en particular, cobrar costos futuros por adelantado. La dificultad está en encontrar, precisamente, una forma de distribuir los costos que haga evidente el costo total.

Función potencial. Esta técnica consiste en definir una función ϕ que depende del objeto que vamos modificando a lo largo de las operaciones. Esta función representa un "ahorro" que vamos haciendo en las operaciones más baratas para poder usarlo en pagar las operaciones más caras a futuro.

Pensemos en un contratista de una obra. Algunos meses tiene más gastos y otros menos. Para tener una interfaz sencilla con su cliente, todos los meses i le cobra lo mismo, $\hat{c}_i = c$. Sin embargo, su costo real, c_i, es variable. En los meses en que $c_i \leq \hat{c}_i$, ahorrará el sobrante $\hat{c}_i - c_i$ en una bolsa llamada ϕ. En los meses en que $c_i > \hat{c}_i$, pagará la diferencia $c_i - \hat{c}_i$ con el ahorro que tiene en la bolsa ϕ. Llamamos ϕ_0 al ahorro inicial con que se comienza la obra y ϕ_i a lo que tiene ahorrado después del mes i. Entonces, se cumple la recurrencia $\phi_i = \phi_{i-1} + \hat{c}_i - c_i$. Como la obra se puede detener en cualquier momento, para asegurarse de no perder dinero el contratista necesita que en todo mes i valga $\phi_i \geq \phi_0$. Desde el punto de vista del cliente, el costo de la obra es constante por mes, $\hat{c}_i$. Esto corresponde al costo amortizado de la operación del contratista, que usa el ahorro para esconder una estructura de costos más complicada.

Formalmente, tenemos entonces una secuencia de costos reales c_i y una función potencial con valor ϕ_i luego de la operación i. Si llamamos $\Delta\phi_i = \phi_i - \phi_{i-1}$, definimos la secuencia de costos amortizados $\hat{c}_i$ como

$$\hat{c}_i \;=\; c_i + \Delta\phi_i.$$

Con esta definición tenemos

$$\begin{aligned}\sum_{i=1}^{n}\hat{c}_i &= \sum_{i=1}^{n}(c_i+\Delta\phi_i),\\ \sum_{i=1}^{n}\hat{c}_i &= \left(\sum_{i=1}^{n}c_i\right)+\phi_n-\phi_0,\\ \sum_{i=1}^{n}\hat{c}_i &\geq \sum_{i=1}^{n}c_i,\end{aligned}$$

donde lo último se cumple si nos aseguramos de que $\phi_n \geq \phi_0$ para toda secuencia de operaciones. Entonces nuestra secuencia de costos amortizados es una cota superior a la secuencia de costos reales.

La dificultad está siempre en definir ϕ adecuadamente para que se mantenga $\phi_n \geq \phi_0$ y sobre todo que los $\hat{c}_i$ resultantes sean fáciles de sumar (constantes, idealmente). Para ello, las operaciones que cuestan mucho deben disminuir el potencial en la misma medida.

En nuestro ejemplo, el potencial podría ser el alto de la pila. Tenemos entonces $\phi_n \geq 0$ y $\phi_0 = 0$. Consideremos lo que ocurre al ejecutar $push(x)$. El costo real es $c_i = 1$. Por otro lado, la pila se hace una unidad más alta, por lo que $\Delta\phi_i = 1$. Esto nos da $\hat{c}_i = c_i + \Delta\phi_i = 2$. Por otro lado, al realizar un $multipop(k)$ tenemos un costo real de $c_i = k$, pero como el alto de la pila decrece en k, tenemos $\Delta\phi_i = -k$, con lo cual el costo amortizado es $\hat{c}_i = c_i + \Delta\phi_i = 0$. Nuevamente, hemos obtenido nuestro costo amortizado de a lo más 2 unidades por operación.

Note que los análisis son válidos si partimos de la pila vacía, pero no si partimos operando con una pila que ya tiene k elementos. En ese caso, una sola operación $multipop(k)$ cuesta $O(k)$ y no $O(1)$. Note que aquí hemos violado algún supuesto hecho en cada una de las tres formas de analizar este problema.

3.2. Incrementar un Número Binario

Seguiremos con un problema también de juguete, aunque algo más complicado y con una aplicación que veremos después. Supongamos que debemos incrementar un número binario de k dígitos, desde 0 hasta $2^k - 1$. Llamemos $n = 2^k$. Consideremos que el costo de incrementar un número es la cantidad de bits que debemos invertir. Como se invierten todos los 1s hasta el primer 0 de derecha a izquierda, el costo de incrementar varía entre 1 y k según el número

que incrementemos. Subrayamos a continuación los bits invertidos para $k = 4$.

$$\begin{array}{cccc} 000\underline{0} & 010\underline{0} & 100\underline{0} & 110\underline{0} \\ 00\underline{01} & 01\underline{01} & 10\underline{01} & 11\underline{01} \\ 001\underline{0} & 011\underline{0} & 101\underline{0} & 111\underline{0} \\ 0\underline{011} & \underline{0111} & 1\underline{011} & 1111 \end{array}$$

El peor caso es, como dijimos, tener que invertir k bits (al pasar de de $2^{k-1} - 1$ a 2^{k-1}), por lo cual la secuencia de las n inversiones cuesta $kn = O(n \log n)$ operaciones. Esto es cierto, pero no ajustado. Veremos que en realidad, se realizan en total menos de $2n = O(n)$ inversiones de bits, es decir, el costo amortizado de incrementar cada número es a lo más 2.

Análisis global. La primera forma de verificar esto es mirar los costos de una forma diferente, que permita sumarlos con más facilidad. Miremos los bits subrayados por columnas y no por filas. Así se verá que el último bit del número cambia siempre, el penúltimo cambia una vez cada 2, el antepenúltimo cambia una vez cada 4, y así. Sumando los bits subrayados por columnas tenemos

$$n + \frac{n}{2} + \frac{n}{4} + \ldots < 2n.$$

Contabilidad de costos. La segunda forma es notar que cada operación realiza exactamente una inversión de la forma $0 \to 1$, y cero o más inversiones $1 \to 0$. Como comenzamos con una secuencia de 0s, todo 1 fue un 0 alguna vez. Podemos entonces cobrarle 2 operaciones a las inversiones $0 \to 1$, y cobrarle cero a las inversiones $1 \to 0$. De este modo, cobramos por adelantado en las inversiones $0 \to 1$ la posible futura inversión $1 \to 0$ de ese bit. Obtenemos entonces una cota superior fácil de sumar: n incrementos cuestan $2n$ inversiones a lo sumo.

Función potencial. Nuestra función potencial podría ser el número de 1s en la secuencia de bits actual. Tenemos entonces $\phi_n \geq 0$ y $\phi_0 = 0$. Consideremos lo que ocurre al incrementar una secuencia que termina con un último 0 y después ℓ 1s. Se invierten el 0 y todos los 1s, por lo que el costo real es $c_i = \ell + 1$. Por otro lado, se pierden ℓ 1s y se gana uno (al convertir el 0 a 1), por lo cual $\Delta\phi_i = -\ell + 1$. Esto nos da $\hat{c}_i = c_i + \Delta\phi_i = 2$, y hemos obtenido nuevamente nuestro costo amortizado.

Nuevamente, los análisis son válidos si partimos de k 0s y realizamos 2^k incrementos o menos, pero no si partimos de otra secuencia de bits. Por ejemplo, si partimos de $2^{k-1} - 1$ y realizamos $n = 1$ operaciones, entonces el costo, real o amortizado, por operación es k, no 2.

3.3. Realocando un Arreglo

Supongamos que vamos leyendo números de un stream y almacenándolos en un arreglo. Como no sabemos cuántos números leeremos, no podemos alocar la memoria definitiva que el arreglo necesitará. Debemos, en cambio, ir realocando áreas de memoria cada vez mayores para el arreglo cada vez que éste se va llenando. Al realocar un arreglo que contiene n elementos, debemos copiar sus n elementos al área nueva de memoria. Consideremos como costo el número total de escrituras a memoria que se requiere a lo largo del proceso.

Si cada vez que se nos llena el arreglo de tamaño n lo realocamos a tamaño $n+1$, usaremos el mínimo posible de memoria, pero el costo total será $\Theta(n^2)$. Lo que se usa es partir con un arreglo de tamaño pequeño (digamos de 1 elemento, para simplificar) y duplicar su tamaño cada vez que se llena. Esto garantiza que en total usamos a lo sumo el doble de la memoria necesaria. Lo que no está claro es cuánto es el costo de insertar un nuevo elemento en el arreglo si realocamos de esta manera.

En términos de peor caso, esta política parece tan mala como la cuadrática: como hay inserciones que nos requieren realocar el arreglo, el costo de peor caso de una inserción son $n + 1$ escrituras, donde n es el número de inserciones anteriores (o el tamaño del arreglo). Necesitamos un análisis amortizado para reflejar el hecho de que, con esta segunda política, estas inserciones tan costosas ocurren muy pocas veces e impactan poco en el costo total.

Análisis global. Podemos notar que los valores de n en los que, al insertar un nuevo elemento, debemos expandir el arreglo (es decir, duplicarlo en tamaño) son las potencias de 2: 1, 2, 4, ..., 2^k. El peor caso es que $n = 2^k + 1$, de modo que no tengamos operaciones baratas después de la última expansión. Además de las n escrituras realizadas para insertar los elementos en el arreglo, hemos copiado $1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1$ elementos a lo largo de todas las expansiones. El costo total es entonces $3 \cdot 2^k < 3n$, y por ende el costo amortizado de una inserción es a lo más de 3 escrituras.

Contabilidad de costos. En vez de cobrar el costo de expandir el arreglo a la operación de inserción, se lo cobraremos a los elementos que se copian, de modo que la operación misma pagará solamente 1 escritura. Cobrarle la operación a los elementos, sin embargo, también nos da un problema, porque a lo largo de todas las expansiones los primeros elementos insertados en el arreglo se copian más veces que los últimos insertados.

Haremos entonces lo siguiente: sólo les cobraremos a los elementos que se copian por primera vez. Cuando se copien $n = 2^k$ elementos, entonces, sólo los $n/2$ de la segunda mitad pagarán la copia, pues estos elementos se están copiando por primera vez. Como alguien tiene que pagar la copia de los $n/2$ de la primera mitad, les cobraremos doble a los de la segunda mitad. Es decir, cada elemento "nuevo" paga por su copia y por la de un elemento "viejo". Luego de copiarse una primera vez, un elemento pasa a ser viejo y a residir en la primera mitad del nuevo arreglo, por lo que no paga nunca más. Esto facilita contabilizar los costos: cada uno de los n elementos puede pagar hasta una vez, a costo 2. Además, cada

una de las n inserciones paga 1, por escribir el elemento en el arreglo. Sumando, tenemos un costo amortizado de a lo más 3 escrituras por inserción.

Función potencial. Sea s el tamaño actual del arreglo, que tiene escritos n elementos, de modo que $n \leq s < 2n$ (excepto al comienzo, en que $n = 0$ y supondremos que partimos con un arreglo de tamaño $s = 1$). Definiremos la función potencial como $\phi = 2n - s$. Al comienzo vale $\phi_0 = -1$, pero luego de la primera inserción siempre vale $\phi_n > 0$. Partiremos la operación de inserción en dos sub-operaciones: la "inserción elemental" y la "expansión". La inserción elemental simplemente agrega el nuevo elemento al arreglo, y sólo puede invocarse cuando hay espacio. La expansión sólo duplica el arreglo y copia los elementos actuales, y se invoca cuando no hay espacio. Así, una inserción consiste de una inserción elemental, a veces precedida de una expansión.

La inserción elemental cuesta $c_i = 1$. Como incrementa n, ocurre que $\Delta\phi_i = 2$. De modo que el costo amortizado de la inserción elemental es $\hat{c_i} = c_i + \Delta\phi_i = 3$. Por otro lado, la expansión teniendo n elementos en el arreglo cuesta $c_i = n$. Se realiza cuando $s = n$ y hace que s pase a ser $2s$. Por lo tanto, $\phi_{i-1} = 2n - s$ y $\phi_i = 2n - 2s$, con lo cual $\Delta\phi_i = -s = -n$. Sumando, tenemos que el costo amortizado de la expansión es $\hat{c_i} = c_i + \Delta\phi_i = 0$. Por lo tanto, el costo amortizado de una inserción es a lo sumo 3.

3.3.1. Parametrizando la solución

Consideremos que, o bien para ahorrar espacio o bien para mejorar el tiempo, decidimos que el arreglo no se duplicará necesariamente, sino que expandirá su tamaño a αn para alguna constante $\alpha > 1$, de modo que a lo sumo usaremos αn celdas de memoria al haber leído n elementos. Nos preguntamos cuál es el costo amortizado.

La forma más fácil de analizar el costo de inserción con este parámetro es modificar la función potencial, que seguirá siendo de la forma $\phi = an - bs$ para constantes adecuadas a y b. Para la inserción elemental tenemos entonces $c_i = 1$ y $\Delta\phi_i = a$, dando un costo amortizado de $\hat{c_i} = 1 + a$. Para la expansión de tamaño $s = n$ a tamaño $s = \alpha n$ tenemos $c_i = n$ y $\Delta\phi_i = -b(\alpha - 1)n$, lo cual nos da $\hat{c_i} = (1 - b(\alpha - 1))n$. Para que esto sea independiente de n (es decir, cero como antes) debemos tener $b = \frac{1}{\alpha - 1}$.

Por otro lado, para que $\phi_n \geq \phi_0$ podemos pedir que $an - bs \geq 0$ (si bien basta $an - bs \geq -b$). Como $s \leq \alpha n$, basta que $a - b\alpha \geq 0$, lo cual significa que podemos elegir $a = \frac{\alpha}{\alpha - 1}$. El costo amortizado, dominado por el de la inserción elemental, es entonces $1 + \frac{\alpha}{\alpha - 1} = \frac{2\alpha - 1}{\alpha - 1}$. Nuestro primer análisis correspondía entonces al caso particular $\alpha = 2$, mientras que ahora podemos ahorrar espacio (aumentando el costo por operación) o reducir el costo por operación (aumentando el espacio). En todo caso, el costo por operación sigue siendo constante si el espacio extra es proporcional a n.

3.3.2. Permitiendo contracciones

Supongamos ahora una situación más compleja en la que también se pueden eliminar elementos en el arreglo. Un elemento eliminado se reemplaza por el que está en último lugar, de modo que el arreglo se almacene en forma compacta. Queremos evitar el estar almacenando demasiada memoria para un arreglo que alguna vez fue grande pero ahora tiene pocos elementos. Para ello, podemos contraer el arreglo cuando queda muy vacío después de un borrado. Esta es la acción contraria a la expansión que se realiza durante la inserción, y supondremos que también cuesta n escrituras.

El primer impulso puede ser que, una vez que el arreglo tenga tamaño $s > \alpha n$, lo contraigamos a tamaño $\frac{s}{\alpha}$. Sin embargo, si volvemos a insertar inmediatamente, el arreglo se volverá a expandir. Por ello, el costo de esta alternativa puede ser muy alto: cada operación costará $\Theta(n)$ escrituras si se elimina un elemento inmediatamente después de expandir, lo que provoca una contracción, luego se vuelve a insertar, lo que provoca otra expansión, luego se vuelve a eliminar, lo que provoca otra contracción, etc.

Para seguir teniendo costo amortizado constante, establecemos que el arreglo se contraerá solamente cuando $s \geq \beta n$, para cierto $\beta > \alpha$, y entonces se contraerá a tamaño γn, para un $1 < \gamma < \beta$. Esto garantiza que la memoria máxima a ser usada es βn. Nos preguntamos ahora cuál es el costo amortizado de una operación.

Esta vez no funciona una función de tipo $\phi = an - bs$. Debemos elegir un esquema algo más complejo: $\phi = |an - bs|$. Tenemos ahora las siguientes operaciones:

- Inserción elemental, a costo $c_i = 1$. El valor de $\Delta\phi_i$ es a lo sumo a, por lo que una cota superior al costo amortizado es $\hat{c}_i \leq 1 + a$.
- Borrado elemental (es decir, sin considerar la contracción), a costo $c_i = 1$. El valor de $\Delta\phi_i$ es también a lo sumo a, por lo que tenemos la misma cota superior $\hat{c}_i \leq 1 + a$.
- Expansión, a costo $c_i = n$. Esta ocurre cuando $n = s$, y pasaremos de $\phi_{i-1} = |an - bs| = |an - bn|$ a $\phi_i = |an - bs\alpha| = |an - bn\alpha|$. Para que nuestro esquema funcione, la expansión debe darse cuando $an - bn\alpha \geq 0$, es decir, que debe valer $a \geq b\alpha$. En este caso, tenemos $\Delta\phi_i = -b(\alpha - 1)n$, y tendremos $\hat{c}_i \leq 0$ siempre que $b \geq \frac{1}{\alpha - 1}$.
- Contracción, a costo $c_i = n$. Esta ocurre cuando $s \geq \beta n$, y pasaremos de $\phi_{i-1} = |an - bs|$ a $\phi_i = |an - b\gamma n|$. Para que nuestro esquema funcione, la contracción debe darse cuando $an - b\gamma n \leq 0$, es decir, que debe valer $a \leq b\gamma$. En este caso, tenemos $\phi_{i-1} = bs - an \geq b\beta n - an$ y $\phi_i = b\gamma n - an$, y entonces $\Delta\phi_i \leq -b(\beta - \gamma)n$. Por lo tanto, tendremos $\hat{c}_i \leq 0$ siempre que $b \geq \frac{1}{\beta - \gamma}$.

Tenemos entonces las condiciones $b\alpha \leq a \leq b\gamma$, $b \geq \frac{1}{\alpha - 1}$ y $b \geq \frac{1}{\beta - \gamma}$. Dado un β fijo (relacionado con la memoria máxima que permitimos desperdiciar), queremos minimizar a (pues el costo amortizado es $1 + a$). El menor a posible es $a = b\alpha$. Usar el menor b posible implica usar el menor γ posible, pues $b \geq \frac{1}{\beta - \gamma}$. Como nuestras desigualdades implican $\gamma \geq \alpha$,

elegimos $\gamma = \alpha$. Las nuevas cotas inferiores son $b \geq \frac{1}{\alpha-1}$ y $b \geq \frac{1}{\beta-\alpha}$, que decrecen y crecen, respectivamente, con α. Por lo tanto, el óptimo se da cuando $\frac{1}{\alpha-1} = \frac{1}{\beta-\alpha}$, es decir, $\alpha = \frac{\beta+1}{2}$. El costo amortizado es entonces $1 + a = 1 + b\alpha = 1 + \frac{\alpha}{\alpha-1} = \frac{2\alpha-1}{\alpha-1} = \frac{2\beta}{\beta-1}$.

Concluyendo, podemos garantizar un uso máximo de βn celdas para almacenar n elementos, permitiendo inserciones y borrados, a un costo amortizado de $\frac{2\beta}{\beta-1}$ por operación. Para lograrlo, cuando insertamos y $s = n$, expandimos el arreglo a $s = \frac{\beta+1}{2}n$, y cuando borramos y $s \geq \beta n$, contraemos el arreglo a $s = \frac{\beta+1}{2}n$. Por ejemplo, podemos elegir $\beta = 3$ para obtener un costo amortizado de $3n$.

3.4. Colas Binomiales

En esta sección veremos una nueva implementación de colas de prioridad. A diferencia de la implementación usando heaps, las colas binomiales permiten *unir* dos heaps de n y m elementos en tiempo $O(\log(n+m))$. La siguiente tabla muestra las complejidades de cada implementación (en términos de $O(\cdot)$).

Implementación	Insert	FindMin	ExtractMin	Heapify	Merge
Heaps	$\log n$	1	$\log n$	n	$n+m$
Cola binomial	$\log n$	1	$\log n$	n	$\log(n+m)$

3.4.1. Estructura

Definimos el *árbol binomial* B_k como una topología, de la siguiente forma:

- B_0 es un árbol formado por un único nodo.
- B_{k+1} es un árbol B_k al que se cuelga de la raíz otro hijo más, que resulta ser la raíz de otro árbol B_k.

Es fácil demostrar por inducción las siguientes propiedades:

- La cantidad de nodos en B_k es 2^k.
- La altura de B_k es $k+1$ (entendiendo que un único nodo tiene altura 1).
- El árbol B_k tiene $\binom{k}{i}$ nodos a profundidad i (entendiendo que la raíz se encuentra a profundidad 0).
- La raíz de B_k tiene k hijos, $B_0, \ldots, B_{k-1}$.

Definimos un *bosque binomial* como un conjunto de árboles binomiales $\{B_{k_1}, \ldots, B_{k_r}\}$ donde ningún par de árboles tiene el mismo tamaño, es decir, $k_i \neq k_j$ para todo $i \neq j$. Tenemos entonces las siguientes propiedades, también fáciles de ver:

- Existe exactamente un bosque binomial de tamaño n para cada $n \geq 0$: la única combinación posible es tomar los B_{k_i} tal que los 1s en la descomposición binaria de n están en las posiciones k_i, partiendo de cero y de derecha a izquierda. Por ejemplo, el único bosque binomial de $n = 5 = 101_2$ nodos es $\{B_2, B_0\}$.
- Un bosque binomial de n nodos tiene a lo más $\lceil \log_2 n \rceil$ árboles binomiales.

Una *cola binomial* para un conjunto de n elementos es un bosque binomial de n nodos donde se almacena un elemento en cada nodo, cumpliendo que si x está almacenado en el padre del nodo donde está almacenado y, entonces $x \leq y$.

3.4.2. Suma de colas

La primitiva crucial en colas binomiales es la *suma*, que dadas dos colas binomiales C_X y C_Y para conjuntos de elementos X e Y entrega una cola binomial C_S para el conjunto $X \cup Y$ (permitimos claves duplicadas, de modo que esta unión no elimina repetidos). La suma procede análogamente a la suma de los números binarios $|X|$ y $|Y|$, partiendo con $C_S = \emptyset$ y considerando los bits en cada posición k de $|X|$ y $|Y|$, desde el bit menos significativo ($k = 0$) al más significativo. Llevaremos también un conjunto T de 0 ó 1 árboles de acarreo, análogo al bit de carry de la suma binaria. Al comenzar tenemos $T = \emptyset$.

1. Si el k-ésimo bit de $|X|$ es 1, mover el árbol B_k de C_X a T.
2. Si el k-ésimo bit de $|Y|$ es 1, mover el árbol B_k de C_Y a T.

Luego, procesamos el T resultante de la siguiente forma:

1. Si $|T| = 0$, no hacer nada para este valor de k.
2. Si $|T| = 1$, mover el árbol B_k de T a C_S, dejando $T = \emptyset$.
3. Si $|T| = 2$, unir los dos árboles B_k y B'_k de T en un árbol B_{k+1}, colgando el que tenga mayor raíz del que tenga menor raíz. El resultado es el nuevo contenido de T para el siguiente valor de k.
4. Si $|T| = 3$, elegir un árbol B_k de los tres y moverlo a C_S. Con los otros dos, proceder como en el punto anterior.

La suma requiere entonces de tiempo $O(\log(|X \cup Y|))$, y deja los conjuntos C_X, C_Y y T vacíos, y el resultado de la suma en C_S.

3.4.3. Operaciones

Consideremos ahora una cola binomial C_S con n elementos, y veamos cómo realizar las operaciones.

Insert. Para insertar un elemento x, creamos una cola binomial $C = \{B_0\}$, con B_0 conteniendo el elemento x, y sumamos las colas C_S y C para formar el nuevo C_S. El elemento queda entonces insertado en tiempo $O(\log n)$.

FindMin. El mínimo de todos los elementos puede calcularse en tiempo $O(\log n)$, mediante recorrer las (a lo más) $\lceil \log_2 n \rceil$ raíces de los árboles del bosque binomial C_S, pues los elementos no-raíces no son menores que las raíces. Para reducir este tiempo a $O(1)$, basta tener precalculado el valor del mínimo: la recorrida de raíces se realiza como postproceso luego de realizar cualquiera de las otras operaciones que modifican C_S, y les agrega sólo un costo adicional de $O(\log n)$.

ExtractMin. Una vez que sabemos que el mínimo es la raíz de un árbol $B_k \in C_S$, sacamos B_k del bosque y eliminamos su raíz. El resultado de eliminar la raíz de B_k es un nuevo bosque binomial formado por los k hijos de la raíz eliminada, $C_N = \{B_0, \ldots, B_{k-1}\}$. Finalmente, sumamos las colas $C_S - \{B_k\}$ y C_N, en tiempo $O(\log n)$, y el resultado es el nuevo C_S.

Heapify. La implementación de heaps tardaría tiempo $O(n \log n)$ en construir un heap mediante inserciones sucesivas, por lo que se diseña para ella un procedimiento especial para realizar esta operación en tiempo $O(n)$. Sin embargo, en una cola binomial obtenemos tiempo $O(n)$ si realizamos n inserciones sucesivas en una cola vacía. La razón está en el análisis de los 2^k incrementos consecutivos en un número de k bits que realizamos al comienzo del capítulo y que está en relación directa con los costos de inserción de esta cola.

Unir. La unión de dos colas binomiales de tamaños m y n se obtiene en tiempo $O(\log(m+n))$ mediante sumarlas. Con un heap clásico, la forma más fácil de unir dos colas de prioridad es concatenar los arreglos e invocar heapify, lo que cuesta tiempo $O(m+n)$.

3.5. Colas de Fibonacci

Las colas de Fibonacci son una variante de las colas binomiales que realizan la inserción y la unión en tiempo constante, mientras que la extracción del mínimo tiene un costo amortizado de $O(\log n)$. Más precisamente, les corresponde la siguiente tabla (donde el asterisco significa tiempo amortizado).

Implementación	Insert	FindMin	ExtractMin	Heapify	Merge
Heaps	$\log n$	1	$\log n$	n	$n+m$
Cola binomial	$\log n$	1	$\log n$	n	$\log(n+m)$
Cola de Fibonacci	1	1	$\log n$ $(*)$	n	1

La principal diferencia con las colas binomiales es que la cola de Fibonacci no es un bosque binomial, sino simplemente un bosque de árboles binomiales unidos en una lista doblemente enlazada. Es decir, la cola de Fibonacci puede tener varios árboles B_k del mismo tamaño. De hecho, una cola de Fibonacci construida mediante n inserciones en una cola vacía no es más que un bosque de n nodos simples. Todo el trabajo de estructurar la cola se realiza al momento de la extracción del mínimo.

Al igual que en la cola binomial, esta cola sabe cuál es la raíz del árbol que contiene el mínimo elemento.

3.5.1. Operaciones

Insert. Para insertar un elemento x en una cola C_S, simplemente se crea un nuevo árbol B_0 conteniendo x y se agrega este B_0 a la lista de árboles de C_S. Además se compara x con el mínimo elemento, para actualizar el mínimo de ser necesario. El tiempo total es $O(1)$.

FindMin. Como siempre conocemos el mínimo elemento, el tiempo es $O(1)$.

Heapify. Se realiza mediante n inserciones, en tiempo $O(n)$.

Merge. Simplemente se unen los dos conjuntos concatenando las dos listas, en tiempo $O(1)$. Además se comparan los dos mínimos, para retener el mínimo global.

ExtractMin. Ésta es la operación más compleja. Aquí recorremos la lista y nos aseguramos de convertirla en un bosque binomial, mediante sumar árboles iguales iterativamente.

Primero eliminamos la raíz del árbol B_k que contiene el mínimo actual (la cual conocemos) y agregamos los hijos $B_0, \ldots, B_{k-1}$ a la lista de árboles de C_S.

Luego, convertimos el bosque de árboles binomiales en un bosque binomial. Para ello, creamos un pequeño arreglo A de $\lceil \log_2 n \rceil$ punteros, donde $A[k]$ apunta a un único árbol B_k si es que tenemos alguno. Inicialmente todos los punteros son nulos. Ahora recorremos la lista. Para cada árbol B_k que encontramos, si $A[k]$ es nulo, asignamos $A[k] \leftarrow B_k$. Si no, unimos B_k con el árbol $A[k]$ (colgando la raíz mayor de la menor) en un único árbol B_{k+1}, dejamos $A[k]$ en nulo y continuamos el proceso con este nuevo árbol B_{k+1}.

Al final de esta operación tenemos un bosque binomial en A, y creamos una lista enlazada con ellos. Ésta es la nueva cola de Fibonacci (en este momento es una cola binomial válida). Sobre las $O(\log n)$ raíces resultantes calculamos el nuevo mínimo y lo recordamos.

3.5.2. Análisis

Para analizar el costo amortizado de las operaciones usaremos la función potencial. Definiremos $\phi = 2\ell + a$, donde ℓ es el largo de la lista de árboles en el bosque y a es la cantidad

de celdas no vacías en el arreglo A que se usa para la operación ExtractMin (se entiende que el arreglo está vacío durante las otras operaciones).

Tenemos $\phi_0 = 0$ al comenzar con la cola vacía. La operación de insertar incrementa ϕ en 2, por lo que su costo amortizado sigue siendo $O(1)$. FindMin no cambia ϕ, por lo que su costo amortizado es igual al costo real, $O(1)$. Heapify es una secuencia de inserciones, por lo que su costo real y amortizado es $O(n)$.

Para la operación Merge, debemos considerar un conjunto de colas, y ϕ se define como la suma de $2\ell + a$ sobre todas las colas. De esa manera, al realizar Merge esta función ϕ global no cambia, y el costo real $O(1)$ es también el costo amortizado.

Nuevamente, la operación compleja es ExtractMin. El primer paso es eliminar la raíz del árbol B_k que contiene el mínimo e insertar sus hijos en la lista. Esto cuesta $c_i = k$ e incrementa el largo de la lista en $k - 1$, con lo cual tenemos $\Delta\phi_i = 2k - 2$ y $\hat{c}_i = 3k - 2 = O(\log n)$ (podríamos tener $c_i = 1$ si representamos los hijos con una lista doblemente enlazada, pero esto no cambia el costo amortizado).

Luego reducimos la lista a un bosque binomial. Consideremos el costo amortizado de procesar cada nuevo árbol B_k. Si $A[k]$ está vacío, movemos el B_k de la lista a $A[k]$, con lo cual tenemos un costo de $c_i = 1$ para moverlo y $\Delta\phi_i = -1$, resultando un costo amortizado de $\hat{c}_i = 0$. Si, en cambio, teníamos un árbol en $A[k]$, entonces lo sacamos de $A[k]$ y lo unimos con el árbol B_k de la lista, reemplazando ese árbol por el árbol unión B_{k+1}, que reemplaza al B_k de la lista. Tenemos un costo de $c_i = 1$ y $\Delta\phi_i = -1$, con lo que nuevamente el costo amortizado es cero. Finalmente, movemos los $a \leq \log n$ árboles no nulos de A a la lista, lo que cuesta a operaciones e incrementa ϕ en a también, sumando $O(\log n)$ al costo amortizado. En total, la operación ExtractMin tiene un costo amortizado de $O(\log n)$.

Note que estamos asignando costo $c_i = 1$ a diversas cantidades constantes de operaciones que realizamos. Se le puede asignar cualquier otro costo constante c y redefinir $\phi = c(2\ell + a)$ para obtener el mismo resultado.

3.6. Union-Find

El algoritmo de Kruskal para encontrar el árbol cobertor mínimo de un grafo crea una cola de prioridad con todas las aristas y parte con un bosque $T = \emptyset$ (visto como conjunto de aristas). Luego va tomando cada arista, y si no forma ciclo en el bosque, la agrega a T, terminando cuando $|T| = n - 1$ y el bosque se ha convertido en un árbol. En un grafo conexo de n nodos y $n - 1 \leq e \leq n^2$ aristas, el algoritmo tiene un peor caso de $O(e \log e) = O(e \log n)$ para manejar la cola de prioridad. Sin embargo, en esta formulación no queda claro cómo determinar si una arista forma ciclo o no.

Si pensamos los árboles de T como clases de equivalencia formadas por nodos, una arista (u, v) forma ciclo sii u y v son nodos del mismo árbol, es decir, están en la misma clase de equivalencia. Cuando no lo están, agregar la arista (u, v) tiene el efecto de unir los dos árboles, es decir, las dos clases de equivalencia. Podemos entonces reexpresar las operaciones

que necesitamos como operaciones que manejan clases de equivalencia:

- Partimos con cada uno de los n elementos formando su propia clase.
- Podemos preguntar si dos elementos pertenecen a la misma clase.
- Podemos unir dos clases.

La interfaz que veremos define un elemento de cada clase, en forma arbitraria pero consistente, como su *representante*. Tenemos entonces las dos operaciones siguientes:

- $Find(v)$ entrega el representante de la clase de equivalencia de v.
- $Union(x, y)$ une las clases de equivalencia representadas por x y por y.

Dos elementos u y v pertenecen entonces a una misma clase sii $Find(u) = Find(v)$, y para unir las clases de dos elementos cualquiera (no necesariamente representantes de clase) u y v realizamos $Union(Find(u), Find(v))$. Con esta interfaz, el algoritmo de Kruskal realiza $O(e)$ operaciones $Find$ y $O(n)$ operaciones $Union$. Veremos primero una solución de tiempo $O(\log n)$ para estas operaciones y luego una mucho mejor, que requiere análisis amortizado. Con estas soluciones, el costo del algoritmo de Kruskal es $O(e \log n)$, dominado por las operaciones en la cola de prioridad.

3.6.1. Solución de tiempo $O(\log n)$

Obviando las soluciones muy elementales, que requieren tiempo $O(n)$ para alguna de las dos operaciones, una solución sencilla es tener n nodos, uno por cada elemento v, y que cada clase de equivalencia sea un árbol formado por los nodos de los elementos participantes. En estos árboles, los hijos apuntan a su padre, y la raíz es el representante de la clase. La operación $Find(v)$ consiste entonces en recorrer los ancestros sucesivos de v hasta llegar a la raíz x, y entonces responder x. La operación $Union(x, y)$, para dos raíces x e y, cuelga x como un hijo más de y (es decir, hace que y sea el padre de x) o vice versa, eligiendo siempre colgar el árbol con menos nodos del árbol con más nodos (cada nodo conoce el tamaño de su subárbol, lo que es fácil de mantener cuando se le agrega otro subárbol como hijo).

Es fácil ver por inducción que, en los árboles resultantes, un árbol de altura r tiene $v \geq 2^r$ nodos. Esto vale para los nodos iniciales, que definimos de altura 0 y que contienen $2^0 = 1$ nodos. Ahora consideremos dos árboles de v_1 y v_2 nodos, y alturas r_1 y r_2, respectivamente. Por hipótesis inductiva se cumple que $v_1 \geq 2^{r_1}$ y $v_2 \geq 2^{r_2}$. Supongamos que $v_1 \leq v_2$, por lo que el primer árbol se cuelga del segundo. El árbol resultante tiene entonces $v = v_1 + v_2$ nodos y su altura es $r = \max(r_1 + 1, r_2)$. Si la altura es $r = r_1 + 1$, la tesis inductiva se cumple porque $v = v_1 + v_2 \geq 2v_1 \geq 2 \cdot 2^{r_1} = 2^{1+r_1} = 2^r$. Si, en cambio, la altura es $r = r_2$, la tesis inductiva se cumple porque $v = v_1 + v_2 \geq v_2 \geq 2^{r_2} = 2^r$.

Como un árbol tiene a lo más n nodos, su altura no puede ser más de $\log_2 n$, por lo cual la operación $Find$ cuesta tiempo $O(\log n)$. Por otro lado, la operación $Union$ es $O(1)$.

3.6.2. Solución de tiempo amortizado $O(\log^* n)$

Cuando se realiza una operación $Find(v)$, se visitan todos los ancestros de v hasta llegar a la raíz x: $v = v_1 \to v_2 \to \ldots \to v_{r-2} \to v_{r-1} \to v_r = x$. Para agilizar las futuras operaciones de $Find$, no nos cuesta nada colgar a todos los nodos del camino, $v_1, \ldots, v_{r-2}$, directamente de $v_r = x$ (por ejemplo, puede hacerse a la vuelta de la recursión). Así, las futuras operaciones $Find(v_i)$ tomarán tiempo $O(1)$. Asimismo, se agilizarán los $Find(u)$ sobre otros descendientes u de algún v_i.

¿Qué impacto tiene esta mejora sobre los tiempos de $Find$? En el peor caso, ninguno, pues si bien una aplicación de $Find$ mejora el tiempo de las siguientes operaciones $Find$, el primer $Find$ puede costar $O(\log n)$ (por ejemplo, si hacemos $n-1$ $Union$ y luego el primer $Find$). Necesitamos entonces realizar un análisis amortizado.

Considere una secuencia S de operaciones $Union$ y $Find$, y llamemos S' a la secuencia S *sin* las operaciones $Find$. Definiremos el *rango* de un nodo v, $r(v)$, como la altura del subárbol luego de realizar las operaciones de S' (o bien, de aplicar S pero sin la mejora que acabamos de describir para $Find$). Hablaremos del rango de los nodos mientras analizamos la secuencia verdadera S, pero debe recordar que $r(v)$ es fijo e independiente del punto de S que estemos considerando.

Una propiedad importante es que, como vimos en la subsección anterior, un nodo de rango r tiene al menos 2^r nodos en su subárbol (el que resulta de aplicar S'). Como, en estos árboles, dos nodos u y v de rango r no pueden descender uno del otro (pues entonces uno sería más alto que el otro), sus subárboles deben ser disjuntos. Por lo tanto, no puede haber más de $\frac{n}{2^r}$ nodos de rango r.

Otra propiedad importante es que, si en algún momento de S, u desciende de v, entonces $r(u) < r(v)$. Esto ocurre porque sólo la operación $Union$ crea nuevas descendencias (al colgar x de y, todo descendiente de x pasa a ser también descendiente de y), mientras que sólo la operación $Find$ destruye descendencias (al colgar todos los v_i directamente de x, los descendientes de v_i dejan de ser descendientes de $v_{i+1}, \ldots, v_{r-1}$). Por lo tanto, en S', donde se han eliminado los $Find$, u también se hará descendiente de v y se mantendrá así hasta el final. Como u desciende de v al final de S', debe ser $r(u) < r(v)$.

Para nuestro análisis, definiremos la función $F(i)$ como $F(0) = 1$ y $F(i) = 2^{F(i-1)}$. Esta función crece muy rápidamente:

i	0	1	2	3	4	5
$F(i)$	1	2	4	16	65536	2^{65536}

Llamaremos $G(n)$ a la inversa de F, $G(n) = \text{mín}\{i,\ F(i) \geq n\}$. La función $G(n)$ también se llama $\log^* n$, y es la cantidad de veces que debemos tomar logaritmo (base 2 en nuestro caso) a n para que sea ≤ 1. En la práctica, vale $G(n) \leq 5$ para cualquier n razonable:

n	0–1	2	3–4	5–16	17–65536	65537–2^{65536}
$G(n)$	0	1	2	3	4	5

Dividiremos a los n nodos en *grupos*: el nodo v pertenecerá al grupo $g(v) = G(r(v))$. Dicho de otro modo, si observamos el bosque que resulta de aplicar S', los nodos de altura 0 y 1 (hojas y padres de sólo hojas) son del grupo $g = 0$, los nodos de altura 2 son del grupo $g = 1$, los de altura 3 y 4 son del grupo $g = 2$, los de altura 5 a 16 son del grupo $g = 3$, etc.

Con estas definiciones ya podemos presentar el análisis amortizado que haremos. Usaremos contabilidad de costos. La operación $Union$ cuesta $O(1)$, por lo que no necesitamos considerarla. Consideraremos que la operación $Find(v)$ cuesta 1 por cada nodo que atravesamos en el camino desde v hasta la raíz x. Este costo, para el análisis, lo repartiremos entre la operación Find misma y los nodos que atravesamos, de la siguiente forma:

- Si, al momento de la operación, el nodo es la raíz x de su árbol, o es hijo de la raíz x, le cobramos a $Find$.
- Si, al momento de la operación, el nodo tiene distinto grupo que su padre, le cobramos a $Find$.
- De otro modo, le cobramos al nodo por el que pasamos.

Note que, cuando recorremos $v = v_1 \to \ldots \to v_r = x$, como cada v_i desciende de v_{i+1}, vimos que debe valer $r(v_i) < r(v_{i+1})$, y por lo tanto $g(v_i) \leq g(v_{i+1})$. Eso significa que cada vez que el grupo de v_i es distinto del de su padre v_{i+1}, el valor del grupo debe aumentar. Como el máximo rango posible es $r = \log_2 n$, los grupos posibles van desde 0 hasta $G(\log_2 n) = G(n) - 1$, y entonces en el camino de v_1 a v_r el valor del grupo puede aumentar sólo $G(n) - 1$ veces. Sumando que $Find$ paga por la raíz $x = v_r$ y su hijo v_{r-1}, tenemos que en cada operación nuestra contabilidad de costos le cobra a $Find$ a lo más $1 + G(n) = O(\log^* n)$.

Debemos ver ahora cuánto les cobramos a los nodos. Note que, como hemos definido la contabilidad, un nodo que paga adquiere un nuevo padre gracias a la mejora que hace $Find$. Este nuevo padre es un ancestro del padre actual, por lo que su rango es estrictamente mayor. Por lo tanto, cada vez que un nodo paga, adquiere un padre de mayor rango. Una vez que adquiere un padre cuyo rango es de un grupo mayor al del nodo, el nodo no pagará nunca más, pues nunca volverá a tener un padre de su mismo grupo (sólo puede seguir adquiriendo padres de mayor y mayor rango).

¿Cuántas veces puede pagar un nodo hasta adquirir un padre de un grupo superior? Si está en el grupo g, y su rango sube sólo de a 1 unidad por vez, puede pagar $F(g) - F(g-1)$ veces hasta que su padre pertenezca al grupo $g + 1$. Digamos para simplificar que los nodos de grupo g pueden pagar $F(g)$ unidades en total. ¿Cuántos nodos hay de grupo g? Digamos que son $N(g)$, con $N(g) = \sum_{r=F(g-1)+1}^{F(g)} M(r)$, donde hay $M(r)$ nodos de rango r. Como vimos que $M(r) \leq \frac{n}{2^r}$, tenemos que

$$N(g) \leq \sum_{r=F(g-1)+1}^{F(g)} \frac{n}{2^r} = \frac{n}{2^{F(g-1)+1}} \times \sum_{r=0}^{F(g)-F(g-1)-1} \frac{1}{2^r} < \frac{2n}{2^{F(g-1)+1}} = \frac{n}{2^{F(g-1)}} = \frac{n}{F(g)}.$$

Es decir, tenemos $N(g) \leq \frac{n}{F(g)}$ nodos del grupo g, y cada uno de ellos paga a lo más $F(g)$ a lo largo de su vida. En total, entre todos los nodos del grupo g pagan a lo más $N(g) \cdot F(g) \leq n$. Como existen $G(n)$ grupos distintos, entre todos los nodos pagan $n \cdot G(n)$. Por lo tanto, si se realizan $\Omega(n)$ operaciones de $Find$, el costo amortizado de $Find$ es $O(\log^* n)$, mientras que su costo de peor caso es $O(\log n)$. El costo de los $Union$ es siempre $O(1)$.

3.7. Splay Trees

Los *splay trees* son árboles binarios de búsqueda que tienen un método distinto de realizar las operaciones, el cual garantiza un costo amortizado de $O(\log n)$ por operación sin necesidad de almacenar información de balanceo como los árboles AVL o Red-Black. Más aún, una secuencia de accesos a los nodos con distintas probabilidades entrega un costo amortizado de $O(H)$, donde H es la entropía de esas probabilidades.

Como la estructura anterior, en el splay tree incluso las operaciones de lectura modifican el árbol. Sus respuestas no cambian, pero se hacen más eficientes gracias a las modificaciones.

3.7.1. Operaciones

La idea principal del splay tree es que el nodo que acaba de accederse debe quedar en la raíz del árbol. Para ello, una vez accedido un nodo x, éste se lleva a la raíz mediante una operación llamada $splay(x)$. Esta operación está formada por una secuencia de rotaciones. Para describirlas, usaremos el formato $z(A, B)$ para indicar un árbol con el elemento z en la raíz, subárbol izquierdo A y subárbol derecho B. Las rotaciones para ir subiendo al nodo x son las siguientes:

- Zig-zig: $z(y(x(A, B), C), D) \rightarrow x(A, y(B, z(C, D)))$.
- Zig-zag: $z(y(A, x(B, C)), D) \rightarrow x(y(A, B), z(C, D))$.
- Zag-zig: $z(A, y(x(B, C), D)) \rightarrow x(z(A, B), y(C, D))$.
- Zag-zag: $z(A, y(B, x(C, D))) \rightarrow x(y(z(A, B), C), D)$.
- Zig: $y(x(A, B), C) \rightarrow x(A, y(B, C))$ (sólo si y es raíz).
- Zag: $y(A, x(B, C)) \rightarrow (x(y(A, B), C)$ (sólo si y es raíz).

Supondremos que las rotaciones dobles cuestan 2 unidades de trabajo y las simples cuestan 1. Las operaciones del árbol se realizan de la siguiente manera:

Buscar. Se busca x como en un árbol binario de búsqueda y luego se hace $splay(x)$. Si x no se encuentra, se hace $splay(x')$, donde x' es el último nodo visitado.

Insertar. Se inserta x como en un árbol binario de búsqueda y luego se hace $splay(x)$.

Borrar. Se borra x como en un árbol binario de búsqueda (es decir, si tiene dos hijos se reemplaza por su sucesor o predecesor in-order), y luego se hace $splay(x')$, donde x' es el padre del nodo finalmente borrado (aquel sucesor o predecesor in-order de x).

3.7.2. Análisis

Note que todas las operaciones del árbol tienen un costo proporcional a la operación *splay* que las sigue. Por lo tanto, podemos concentrarnos en el costo de esta operación (si bien luego consideraremos la modificación que hacen en el árbol la inserción y el borrado).

Para analizar esta operación en forma amortizada definiremos una función potencial. Sea $S_i(x)$ el subárbol con raíz x luego de la operación i, y sea $s_i(x) = |S_i(x)|$ su número de nodos. Finalmente, sea $r_i(x) = \log_2 s_i(x)$ el *rango* de x luego de la operación i. La función potencial del árbol T, visto como un conjunto de nodos, es entonces

$$\phi_i = \sum_{x \in T} r_i(x).$$

Analicemos cómo cambia $\Delta\phi_i$ luego de realizar las rotaciones.

Zig-zig y zag-zag. Los únicos nodos cuyos rangos se modifican son los de x, y, y z. Por lo tanto,

$$\Delta\phi_i = (r_i(x) - r_{i-1}(x)) + (r_i(y) - r_{i-1}(y)) + (r_i(z) - r_{i-1}(z)).$$

Note también los siguientes hechos simples: (1) $r_i(x) = r_{i-1}(z)$; (2) $r_{i-1}(y) \geq r_{i-1}(x)$; (3) $r_i(y) \leq r_i(x)$. El costo real de hacer un zig-zig es $c_i = 2$, mientras que el costo amortizado es

$$\begin{aligned}
\hat{c_i} = c_i + \Delta\phi_i &= 2 + (r_i(x) - r_{i-1}(x)) + (r_i(y) - r_{i-1}(y)) + (r_i(z) - r_{i-1}(z)) \\
&= 2 + [r_i(x) - r_{i-1}(z)] + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y) \\
&\leq 2 + r_i(x) + r_i(z) - r_{i-1}(x) - r_{i-1}(x) \\
&= 2 + r_i(x) + r_i(z) - 2r_{i-1}(x),
\end{aligned}$$

donde en la desigualdad usamos los tres hechos simples mencionados.

Vamos a usar la siguiente propiedad, que se deduce de la concavidad del logaritmo, pero

igual mostramos su deducción:

$$\begin{aligned} 0 &\leq (a-b)^2, \\ 2ab &\leq a^2+b^2, \\ 4ab &\leq (a+b)^2, \\ \log_2(4ab) &\leq \log_2((a+b)^2), \\ \log_2(ab) &\leq 2\log_2(a+b)-2, \\ \log_2 a + \log_2 b &\leq 2\log_2(a+b)-2. \end{aligned}$$

Usaremos esta propiedad y el hecho simple de que $s_{i-1}(x) + s_i(z) \leq s_i(x)$ para obtener lo siguiente:

$$\begin{aligned} r_{i-1}(x) + r_i(z) &= \log_2 s_{i-1}(x) + \log_2 s_i(z) \leq 2\log_2(s_{i-1}(x)+s_i(z)) - 2 \\ &\leq 2\log_2 s_i(x) - 2 = 2r_i(x) - 2. \end{aligned}$$

Por lo tanto, podemos reemplazar $r_i(z)$ por $2r_i(x) - r_{i-1}(x) - 2$ en nuestra ecuación de $\hat{c}_i$ para obtener

$$\hat{c}_i \leq 3(r_i(x) - r_{i-1}(x)).$$

Zig-zag y zag-zig. Partimos de la misma forma que antes y luego acotamos:

$$\begin{aligned} \hat{c}_i = c_i + \Delta\phi_i &= 2 + (r_i(x) - r_{i-1}(x)) + (r_i(y) - r_{i-1}(y)) + (r_i(z) - r_{i-1}(z)) \\ &= 2 + [r_i(x) - r_{i-1}(z)] + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y) \\ &\leq 2 + r_i(y) + r_i(z) - 2r_{i-1}(x), \end{aligned}$$

donde en la desigualdad usamos que $r_i(x) = r_{i-1}(z)$ y que $r_{i-1}(y) \geq r_{i-1}(x)$. Ahora volvemos a usar que $\log_2 a + \log_2 b \leq 2\log_2(a+b) - 2$ y que $s_i(y) + s_i(z) \leq s_i(x)$ para acotar

$$\begin{aligned} r_i(y) + r_i(z) &= \log_2 s_i(y) + \log_2 s_i(z) \leq 2\log_2(s_i(y)+s_i(z)) - 2 \\ &\leq 2\log_2 s_i(x) - 2 = 2r_i(x) - 2. \end{aligned}$$

Sustituyendo en la fórmula de $\hat{c}_i$ tenemos entonces

$$\hat{c}_i \leq 2(r_i(x) - r_{i-1}(x)) \leq 3(r_i(x) - r_{i-1}(x)).$$

Zig y zag. En estas operaciones tenemos $c_i = 1$. Como $r_i(y) \leq r_{i-1}(y)$ y $r_i(x) \geq r_{i-1}(x)$,

$$\begin{aligned} \hat{c}_i &= c_i + (r_i(x) - r_{i-1}(x)) + (r_i(y) - r_{i-1}(y)) \\ &\leq 1 + r_i(x) - r_{i-1}(x) \\ &\leq 1 + 3(r_i(x) - r_{i-1}(x)). \end{aligned}$$

Splay. Una operación *splay* se compone de una secuencia de m rotaciones dobles consecutivas, posiblemente terminadas por una simple, todas aplicadas sobre el mismo nodo x. Su costo amortizado es entonces

$$\hat{c} \;\leq\; 1 + \sum_{i=1}^{m} 3(r_i(x) - r_{i-1}(x)) \;=\; 1 + 3(r_m(x) - r_0(x)) \;\leq\; 1 + 3\log_2 n,$$

donde la última desigualdad se obtiene notando que $r_m(x) = \log_2 n$ porque x termina siendo la raíz, y despreciando $r_0(x) \geq 0$.

Operaciones del árbol. La cota de splay implica directamente una cota amortizada de $O(\log n)$ para las operaciones de búsqueda exitosa sobre un splay tree. Lo mismo ocurre con la búsqueda infructuosa, la inserción y el borrado, ya que hemos hecho que su costo sea proporcional a la operación de splay correspondiente. La única consideración final que necesitamos es que la inserción, al agregar un elemento x, puede incrementar el potencial ϕ, lo cual debe ser absorbido por el costo amortizado de la inserción. La diferencia de potencial que produce la inserción de una hoja x, siendo $y_1, \ldots, y_m$ los nodos de su camino hacia la raíz y $s(y_j)$ sus tamaños, son

$$\begin{aligned}
\Delta\phi &= \sum_{j=1}^{m} (\log_2(s(y_j)+1) - \log_2 s(y_j)) \\
&\leq (\log_2(s(y_m)+1) - \log_2 s(y_m)) + \sum_{j=1}^{m-1} (\log_2 s(y_{j+1}) - \log_2 s(y_j)) \\
&= \log_2(s(y_m)+1) - \log_2 s(y_m) + \log_2 s(y_m) - \log_2 s(y_1) \\
&\leq \log_2(n+1) - 1 \\
&\leq \log_2 n,
\end{aligned}$$

donde usamos que, como y_{j+1} es el padre de y_j, debe valer $s(y_{j+1}) \geq s(y_j) + 1$. El incremento de potencial de la inserción es también $O(\log n)$, lo que completa el análisis.

3.7.3. Búsquedas con distintas probabilidades de acceso

El mayor interés de los splay trees es que se acercan a los árboles óptimos que diseñamos en el capítulo de cotas inferiores cuando teníamos probabilidades de acceso conocidas, pero sin necesidad de ningún preprocesamiento. Demostraremos que si realizamos m búsquedas sobre un splay tree T, donde el elemento x es buscado $q(x)$ veces (por lo tanto $\sum_{x \in T} q(x) = m$), entonces el costo total de las búsquedas es

$$O\left(m + \sum_{x \in T} q(x) \log \frac{m}{q(x)}\right).$$

Para ello, reusaremos el análisis ya hecho y sólo cambiaremos la noción de tamaño de un árbol. Definiremos ahora el *peso* de un nodo como la probabilidad de ser buscado, $w(x) = \frac{q(x)}{m}$, y definiremos el tamaño de un subárbol con raíz x como la probabilidad de que la búsqueda pase por el nodo x, es decir,

$$s_i(x) \quad = \quad \sum_{y \in S_i(x)} w(y).$$

Note que $s_i(x)$ es un número entre 0 y 1, por lo que $r_i(x)$ es negativo, pero aún el logaritmo es monótonamente creciente. Todo el análisis realizado anteriormente vale, pues lo único que utilizamos sobre $s_i(x)$ fue que $s_i(\cdot)$ (y $r_i(\cdot)$) es mayor en un subárbol que en un subconjunto disjunto de sus subárboles. Tenemos entonces que el costo amortizado de $splay(x)$ es

$$\begin{aligned}
\hat{c}(x) &\leq 1 + 3(r_m(x) - r_0(x)) &&= 1 + 3(\log_2 s_m(x) - \log_2 s_0(x)) \\
&= 1 + 3(\log_2 1 - \log_2 s_0(x)) &&= 1 + 3 \log \frac{1}{s_0(x)} \\
&\leq 1 + 3 \log \frac{1}{w(x)} &&= 1 + 3 \log \frac{m}{q(x)}.
\end{aligned}$$

En la segunda línea usamos que, al final de la operación, x está en la raíz, por lo cual $s_m(x) = 1$. En la tercera línea usamos que, sin importar dónde estuviera x en el árbol antes de empezar la operación, tendremos $s_0(x) \geq w(x)$.

Sabemos que realizamos $splay(x)$ $q(x)$ veces, de modo que sumando sobre todos los x obtenemos el costo prometido. Note que esto implica un costo amortizado de $O(1 + H)$ por operación, donde H es la entropía de las probabilidades de acceso a los elementos.

Los splay trees tienen otras propiedades interesantes que se pueden demostrar de forma similar, variando la definición de $s(x)$.

3.8. Ficha Resumen

Técnicas:

- Análisis global.
- Contabilidad de costos.
- Función potencial.

Ejemplos relevantes:

- Realocar arreglo: αn memoria para n elementos, a costo amortizado de inserción $\frac{2\alpha-1}{\alpha-1}$. Permitiendo borrados, βn memoria a costo amortizado de operación $\frac{2\beta}{\beta-1}$.

- Colas binomiales: $O(\log n)$ para insertar, extraer mínimo y unir, $O(1)$ para ver el mínimo y $O(n)$ para construir a partir de n elementos.
- Colas de Fibonacci: $O(1)$ para insertar, ver el mínimo y unir, $O(n)$ para construir a partir de n elementos, y $O(\log n)$ amortizado para extraer mínimo.
- Union-Find: $O(1)$ para union y $O(\log^* n)$ amortizado para find.
- Splay trees: $O(\log n)$ amortizado para insertar, borrar y buscar. $O(H)$ para una secuencia de búsquedas, siendo H la entropía de la secuencia de elementos accedidos.

3.9. Material Suplementario

Cormen et al. [CLRS01, cap. 17] presentan las tres técnicas de análisis amortizado usando los ejemplos del multipop y el incremento de números binarios. También presentan el caso de los arreglos que se expanden y contraen, si bien usan una función potencial algo distinta. Lee et al. [LTCT05] también usan el multipop para introducir el análisis amortizado usando función potencial. Mehlhorn y Sanders [MS08, sec. 3.3] también describen el problema de los arreglos y de los números binarios, y los toman como punto de partida para explicar las técnicas de análisis amortizado en bastante profundidad. En particular, demuestran que la función potencial es suficiente para cualquier análisis amortizado (si bien puede que no siempre sea la técnica más intuitiva para usar).

Cormen et al. [CLRS01, cap. 19] describen las colas binomiales y también las de Fibonacci [CLRS01, cap. 20], si bien nuestra descripción de estas últimas es más simple, porque ellos permiten una operación que decrementa el valor de una clave, lo que requiere una función potencial más complicada (esta operación es importante para mostrar que el algoritmo de Dijkstra sobre un grafo de n nodos y e aristas se puede ejecutar en tiempo $O(e + n \log n)$). Weiss también describe las colas binomiales [Wei95, sec. 6.8 y 11.2] y las de Fibonacci [Wei95, sec. 11.4]. Asimismo Weiss [Wei95, sec. 6.6] describe la *leftist heap*, cuya rama más izquierda es de largo $O(\log n)$. La leftist heap reduce todas las operaciones a la unión de dos heaps, que se ejecuta en tiempo $O(\log n)$. También describe las *skew heaps* [Wei95, sec. 6.7 y 11.3], una variante amortizada de las leftist heaps y muy simples de implementar. Las skew heaps también se analizan en Lee et al. [LTCT05, sec. 10.2].

Nuestra descripción de Union-Find está sacada de Aho et al. [AHU74], y Weiss [Wei95, cap. 8] la presenta de forma muy similar. Cormen et al. [CLRS01, cap. 21] también la describen en detalle, pero usan un análisis de función potencial bastante más complicado, que les entrega un mejor resultado: n operaciones cuestan $O(n \cdot \alpha(n))$. Esta $\alpha(n)$ es la inversa de una función que crece aún más rápidamente que nuestra $F(i)$, por lo que $\alpha(n) \leq 4$ para todo valor práctico de n. Lee et al. [LTCT05, sec. 10.6] realizan un análisis similarmente complejo. Aho et al. [AHU83], Mehlhorn y Sanders [MS08, sec. 11.4], Kleinberg y Tardos

[KT06, sec. 4.6] y Levitin [Lev07, sec. 9.2] también describen esta implementación de Union-Find, pero sin análisis.

Weiss [Wei95, sec. 4.5, 11.5 y 12.1] describe los splay trees con bastante detalle, pero sólo muestra la cota de $O(\log n)$ amortizado. En las referencias online se encuentran los casos de distintas probabilidades de acceso (llamado static optimality) y otros.

Otros casos interesantes de análisis amortizado se dan en varios tipos de árboles balanceados y colas de prioridad [CLRS01, probl. 17-3] [MS08, sec. 7.4] [LTCT05, sec. 10.3 y 10.5], algoritmos para flujo en redes [CLRS01, sec. 26.4] [KT06, sec. 7.4] y para scheduling en discos [LTCT05, sec. 10.7]. Veremos otros casos de análisis amortizado en el capítulo de competitividad.

Otras fuentes online de interés:

- www.cs.princeton.edu/ wayne/cs423/lectures/amortized-4up.pdf
- www3.cs.stonybrook.edu/~rezaul/Fall-2012/CSE548/CSE548-lectures-10-11.pdf
- courses.cs.washington.edu/courses/cse332/10sp/lectures/lecture21.pdf
- www.cs.unm.edu/~saia/classes/561-f09/lec/lec8.pdf
- jeffe.cs.illinois.edu/teaching/algorithms/2009/notes/08-amortize.pdf
- www.cs.cmu.edu/afs/cs/academic/class/15451-s07/www/lecture_notes/lect0206.pdf
- www.cs.cmu.edu/~rjsimmon/15122-f14/lec/12-ubarrays.pdf
- www.cs.duke.edu/courses/fall12/compsci330/restricted/lectures/LectureAmortizedAnalysis.pdf
- www.cs.princeton.edu/~fiebrink/423/AmortizedAnalysisExplained_Fiebrink.pdf
- www.ibr.cs.tu-bs.de/courses/ss13/na/skript/Fib2.pdf
- users.info.uvt.ro/~mmarin/lectures/ADS/ADS-L9-10.pdf
- ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-854j-advanced-algorithms-fall-2008/lecture-notes/lec6.pdf
- web.stanford.edu/class/cs166/lectures/10/Small10.pdf
- www.cs.cmu.edu/ ckingsf/bioinfo-lectures/splaytrees.pdf
- jeffe.cs.illinois.edu/teaching/algorithms/notes/16-scapegoat-splay.pdf
- www.youtube.com/watch?v=3MpzavN3Mco
- www.youtube.com/watch?v=qh5lSHCBiRs

Capítulo 4

Universos Discretos y Finitos

Los métodos y cotas inferiores que hemos visto para ordenamiento y búsqueda aplican a algoritmos que proceden por comparaciones. Estos tienen la gracia de que son lo más general posible, es decir, se pueden usar en cualquier conjunto de objetos que tengan un orden total. En muchos casos, sin embargo, los objetos que manejamos son de tipos particulares, como enteros en un rango acotado o strings, y en esos casos es posible ordenar y buscar de otras formas, sin usar comparaciones. En estos casos, las cotas inferiores de $\Omega(n \log n)$ para ordenar o de $\Omega(\log n)$ para buscar ya no aplican, y efectivamente es posible diseñar algoritmos y estructuras de datos más eficientes. En este capítulo veremos formas más eficientes de ordenar enteros en un rango acotado, de buscar en un universo acotado de enteros, y de ordenar y buscar en strings. Veremos también estructuras especiales para buscar en texto.

4.1. Ordenando en Tiempo Lineal

Veremos que, si tenemos n elementos en un universo entero $[1..u]$, podemos ordenarlos en tiempo $O(n \log_n u)$. En particular, esto es $O(n)$ cuando el tamaño del universo es de la forma $u = O(n^c)$ para alguna constante c.

Comenzaremos con un método de ordenamiento muy simple de tiempo $O(n+u)$, llamado counting sort. Como este método no permite distinguir dos elementos distintos con una misma clave, seguiremos con bucket sort, que es una variante algo más sofisticada con la misma complejidad. El bucket sort será usado entonces para construir el radix sort, que obtendrá finalmente las cotas prometidas. Finalmente, usaremos el bucket sort para ordenar strings en tiempo lineal.

4.1.1. Counting sort

Supongamos que tenemos que ordenar $A[1..n]$, donde cada $A[i] \in [1..u]$. El counting sort comienza inicializando un arreglo de contadores $C[1..u]$, todos en cero, $C[j] \leftarrow 0$. Luego

recorre A, incrementando el contador correspondiente, $C[A[i]] \leftarrow C[A[i]] + 1$. Finalmente, recorre C, escribiendo (en A) $C[j]$ copias del valor j, para $j \in [1..u]$.

Esta técnica puede aplicarse cuando los elementos son solamente claves, es decir, no tienen información satélite asociada. El tiempo de las tres pasadas es $O(n + u)$, por lo cual es conveniente solamente cuando el universo es pequeño (por ejemplo, $u = O(n)$). Note también que necesita espacio extra para u contadores, lo cual puede ser significativo incluso si $u = O(n)$ con una constante no muy pequeña.

4.1.2. Bucket sort

Para el caso en que los elementos tengan información satélite asociada, el bucket sort obtiene la misma complejidad que el counting sort, y de hecho comienza de la misma forma, calculando el arreglo C de contadores. Luego, supondremos que tenemos también la celda $C[0] = 1$, y convertiremos C en un arreglo de *punteros*, mediante recorrerlo de izquierda a derecha y calcular $C[j] \leftarrow C[j-1] + C[j]$, para $j \in [1..u]$. La idea es que ahora $C[j-1]$ almacena la posición donde deben empezar a escribirse las copias del valor j en A.

El output del algoritmo es un nuevo arreglo $B[1..n]$, que contendrá los valores de A ordenados. Para llenarlo, volvemos a recorrer A de izquierda a derecha, copiando $B[C[A[i]-1]] \leftarrow A[i]$ y luego incrementando el puntero $C[A[i]-1]$ para la siguiente ocurrencia del valor $A[i]$. Se entiende que, al copiar $A[i]$ en B, se copia la clave y la información satélite.

Note que el bucket sort es estable: dos elementos con la misma clave mantienen su orden original en A. Esto es importante para usarlo en el radix sort. Por otro lado, note que el bucket sort requiere espacio para almacenar B y C, más que el counting sort.

4.1.3. Radix sort

El radix sort realiza una serie de rondas de bucket sort sobre los datos, ordenándolos progresivamente de los bits menos a los más significativos. Suponga que primero ordenamos $A[1..n]$ usando como clave el bit más bajo. Como el bit es la clave, el universo sólo tiene dos valores, lo que lo hace un caso fácil de bucket sort, que tomará tiempo $O(n + 2) = O(n)$. Una vez ordenado por el bit más bajo, volvemos a ordenar A por el segundo bit más bajo. Al ser estable el bucket sort, los elementos que tengan su segundo bit más bajo iguales se mantendrán ordenados por el bit más bajo, resultado de la ronda previa de ordenamiento.

Si continuamos hasta ordenar por el bit más alto, el arreglo $A[1..n]$ quedará finalmente ordenado. Como los números en A se representan en $\lceil \log_2 u \rceil$ bits, esta técnica requiere tiempo $O(n \log u)$, lo cual en principio no es tan bueno.

Sin embargo, no necesitamos ordenar de a un bit por vez. En cambio, podemos ordenar de a $k = \lfloor \log_2 n \rfloor$ bits en cada ronda. Es decir, consideraremos primero los k bits más bajos, luego los siguientes k bits, etc. Como las claves son ahora de k bits, el universo es de tamaño $2^k \leq n$, y el bucket sort aún requerirá tiempo $O(n + 2^k) = O(n)$ en cada ronda. El total de rondas es $\frac{\lceil \log_2 u \rceil}{k} = O(\frac{\log u}{\log n}) = O(\log_n u)$, y por ende el tiempo total es $O(n \log_n u)$.

El espacio requerido es también $O(n)$, que se reparte entre el arreglo $B[1..n]$ y los contadores $C[1..2^k]$. Es posible reducir significativamente el espacio de C mediante reducir k. Por ejemplo, ordenando de a $k/2$ bits, realizaremos el doble de rondas (lo que no alterará la complejidad) y el espacio para C será solamente $O(\sqrt{n})$. Este ahorro de espacio puede redundar además en una mayor eficiencia del algoritmo (a pesar de realizar más rondas) dado que mejorará notablemente su localidad de referencia.

4.1.4. Ordenando strings

Suponga que tiene n strings de largo m, dando un tamaño total $N = nm$. Supondremos que los strings usan un alfabeto de tamaño constante σ. Un método clásico de ordenamiento aplicado a los strings requiere tiempo $O(mn \log n) = O(N \log n)$, dado que cada comparación de dos strings puede requerir tiempo $O(m)$.

En cambio, podemos ordenar los strings en tiempo $O(N)$ mediante realizar rondas sucesivas de bucket sort, al igual que radix sort. Equivalentemente a ver los strings como números en base σ, ordenaremos los strings por su último carácter, luego por su penúltimo carácter, etc. Al final, los strings quedarán ordenados lexicográficamente, es decir por su primer carácter, los que coincidan en su primer carácter por el segundo, los que coincidan en sus dos primeros caracteres por el tercero, etc. Cada ronda de bucket sort cuesta tiempo $O(n+\sigma) = O(n)$, y se realizan m rondas, con lo que el tiempo total es $O(mn) = O(N)$. Note que el arreglo A contendrá punteros a los strings, de modo que cada vez que se reescribe un $A[i]$ en B lo único que se copia es el puntero al string, en tiempo constante.

Consideremos ahora el caso en que los n strings tienen distintos largos $m_i \geq 1$, sumando un largo total de $N = \sum_{i=1}^{n} m_i$. Aún es posible ordenarlos en tiempo $O(N)$. Lo que haremos será partir ordenando los strings por su largo, de más corto a más largo. Luego, partiremos del último carácter de los strings más largos, e iremos considerando los caracteres previos, incorporando nuevos strings al conjunto cuando sus largos m_i se van alcanzando en el proceso.

Más en detalle, usaremos bucket sort para ordenar los strings por largo (es decir, la clave es su largo). Como los largos no pueden exceder N, el bucket sort requerirá tiempo $O(n + N) = O(N)$ (en la práctica, bastante menos). Consideremos ahora los n strings $S_1, \ldots, S_n$, ordenados de más corto a más largo. Partiremos del largo máximo, $m = m_n$, y un cursor en $p = n + 1$. Ahora decrementaremos p hasta el mínimo valor posible tal que $m_i \geq m$ para todo $p \leq i \leq n$ (es decir, para incluir a todos los strings de largo máximo, pues podría haber más de uno). Realizaremos una ronda de bucket sort con el carácter m-ésimo de los strings $A[p..n] = S_p, \ldots, S_n$. Luego decrementaremos $m \leftarrow m - 1$ y volveremos a decrementar p para incluir los posibles strings del nuevo largo m. Estos nuevos elementos se incorporarán al conjunto $A[p..n]$, siendo considerados inicialmente menores que los del conjunto anterior. Esto es compatible con el orden lexicográfico, donde un string que es un prefijo de otro se considera lexicográficamente menor. Continuaremos las rondas hasta procesar el carácter $m = 1$ de todos los strings.

Es fácil ver que el bucket sort de la ronda m trabaja $O(k_m + \sigma) = O(k_m)$, donde k_m es

el número de strings de largo $\geq m$. Como $\sum_{m=1}^{m_n} k_m = N$, el tiempo total es $O(N)$. Esto incluye también el tiempo de decrementar p, que suma $O(n)$ en total.

4.2. Predecesor en Tiempo Loglogarítmico

Dado un conjunto de elementos $X = \{x_1, \ldots, x_n\}$, con $x_i < x_{i+1}$, el *predecesor* de y en X es el máximo $x_i \leq y$ (definiremos que el predecesor es $-\infty$ cuando $y < x_1$). Note que, si $y \in X$, entonces su predecesor es el mismo y, y si no, es un elemento estrictamente menor. Por ello, una estructura que implemente consultas de predecesor puede usarse como diccionario (es decir, como una estructura en la que se puede insertar, borrar, y buscar si un elemento está o no), pero es en realidad más potente. Puede usarse en cualquier caso en que quiera encontrarse el valor más cercano a uno dado. En particular, estas operaciones permiten simular una cola de prioridad que extraiga el máximo como el predecesor de $+\infty$ (obviamente podemos tener una cola de prioridad que extraiga el mínimo mediante implementar una estructura de sucesor en vez de predecesor, o cambiando el signo de todos los números).

Si procedemos por comparaciones, tanto un diccionario como una estructura de predecesor se pueden implementar en tiempo $O(\log n)$ para todas las operaciones usando un árbol balanceado, y ese tiempo calza con la cota inferior de $\Omega(\log n)$ de Teoría de la Información que vimos. Es decir, los dos problemas son equivalentes en el modelo de comparaciones.

Fuera del modelo de comparaciones, sin embargo, los problemas difieren. Usando hashing, podemos implementar diccionarios en tiempo esperado $O(1)$, e incluso en $O(1)$ peor caso si no permitimos modificaciones al conjunto (usando hashing perfecto, que veremos más adelante). Esto no es posible para el problema del predecesor.

En el caso de universos discretos los problemas también son diferentes. Si los x_i pertenecen a un universo discreto $[1..u]$, y nos permitimos usar espacio proporcional a u, entonces un simple arreglo de u bits implementa un diccionario con todas las operaciones en tiempo $O(1)$. El problema del predecesor se puede implementar en tiempo $O(1)$ con un arreglo de u enteros que almacenan todas las respuestas, pero esto no permite modificar el conjunto X.

El problema del predecesor tiene una cota inferior de $\Omega(\log \log_w \frac{u}{n})$, para una palabra de máquina de w bits. Esta cota es válida tanto si se permiten modificaciones a X como si no se permiten pero se limita el espacio a polinomial en n. Es válida incluso si nos referimos al caso promedio, y si nos conformamos con tener sólo una probabilidad fija de responder correctamente.

En esta sección veremos una estructura que usa espacio $O(u)$ y tiempo $O(\log \log u)$ para todas las operaciones, lo que está bastante cerca de la cota inferior. Note que esto implica que podemos tener una cola de prioridad con tiempos $O(\log \log u)$ para todas las operaciones, y que podemos ordenar en tiempo $O(n \log \log u)$.

Existen variantes más complejas de la estructura que usan espacio $O(n)$ pero no permiten modificaciones a X, o bien los tiempos de operación son esperados y no de peor caso. Veremos un ejemplo de ellas.

4.2.1. El van Emde Boas tree

El van Emde Boas (vEB) tree particiona recursivamente el universo en subuniversos fijos (en vez de particionar los datos, como los árboles clásicos). Además, no lo particiona en una cantidad constante de hijos, que llevaría a una altura logarítmica, sino en una cantidad mayor, que hace que su altura sea loglogarítmica.

Un vEB-tree para un universo $[0..u-1]$, $vEB(u)$, tiene los siguientes componentes:

- Enteros min, max, $size$, que indican, respectivamente, el valor mínimo y máximo y la cantidad de elementos en el árbol.
- Un array $bottom[0..\sqrt{u}-1]$ de subárboles de tipo $vEB(\sqrt{u})$, de modo que $bottom[i]$ contiene los elementos que pertenecen al subuniverso $[i\cdot\sqrt{u}..(i+1)\cdot\sqrt{u}-1]$.
- Un subárbol top de tipo $vEB(\sqrt{u})$, donde el elemento i está en top sii $bottom[i]$ no está vacío.

¿Hasta dónde sigue la descomposición recursiva? Para garantizar tiempos $O(\log\log u)$, podemos detenernos cuando el tamaño del subuniverso es $O(\log^c u)$, para alguna constante c, de modo que podamos almacenar en un árbol balanceado los elementos que caen en ese subuniverso. Como este árbol no puede tener más de $O(\log^c u)$ elementos, el tiempo de operación en él sumará $O(c\log\log u)=O(\log\log u)$ al tiempo total.

¿Qué altura tiene el vEB tree? Supongamos para simplificar que continuamos con la descomposición recursiva hasta que el subuniverso es de tamaño 2. La altura obedece entonces a la recurrencia $H(u)=1+H(\sqrt{u})$ y $H(2)=0$, cuya solución es $\log_2\log_2 u=O(\log\log u)$.

¿Qué espacio requiere esta estructura? Esto corresponde a la recurrencia $S(u)=3+(\sqrt{u}+1)\cdot S(\sqrt{u})$ y $S(2)=3$. Podemos mostrar por inducción que $S(u)\le 6u-9=O(u)$, mediante comprobar el caso base y sustituir para el caso inductivo. Si, en cambio, nos detenemos en un subuniverso de tamaño $O(\log^c u)$, entonces los árboles balanceados sumarán $O(n)$, pues en total almacenan todos los elementos, y la estructura principal sumará $O(\frac{u}{\log^c u})$. Es decir, podemos dividir la influencia de la estructura global por cualquier polylog, ocupando espacio $O(n+\frac{u}{\log^c u})$, si pagamos $O(c\log\log u)$ por las búsquedas.

4.2.2. Búsquedas

Consideremos el problema de encontrar el predecesor de y en X, que está almacenado en un vEB tree. Lo primero es descomponer $y=a\sqrt{u}+b$, con $0\le b<\sqrt{u}$. Esto implica que y pertenece al subuniverso a, dentro del cual su valor relativo es b. Esta descomposición es una división, pero puede hacerse más eficientemente usando operaciones de bits si u es de la forma $u=2^{2^k}$. En este caso, b son los 2^{k-1} bits más bajos de y, mientras que a son los 2^{k-1} bits más altos (en C, $a=y>>2^{k-1}$ y $b=y\,\&\,((1<<2^{k-1})-1)$).

Una vez que sabemos que y está en el subuniverso a, podemos determinar si su predecesor también se halla ahí. Esto ocurre sii $bottom[a].size>0$ y $bottom[a].min\le b$. Si es ese el

caso, entonces podemos recursivamente encontrar el predecesor b' de b en el vEB $bottom[a]$, y luego responder traduciendo b' del subuniverso a al universo global, $a\sqrt{u} + b'$.

Si, en cambio, vemos que b no tiene un predecesor en $bottom[a]$, entonces el predecesor de y debe encontrarse en el subuniverso no vacío más cercano a la izquierda de a. Podemos encontrar este subuniverso, $a' < a$, mediante calcular el predecesor de $a-1$ en el vEB top, que contiene precisamente los subuniversos no vacíos. Si no existe tal a', entonces tampoco existe el predecesor de y. Si existe, entonces el predecesor de y es el máximo elemento almacenado en $bottom[a']$, es decir, la respuesta es $a'\sqrt{u} + bottom[a'].max$.

El tiempo de búsqueda resulta ser $O(\log \log u)$ si notamos que, en cada universo de tamaño u, reducimos el problema a una búsqueda o bien en un $bottom[a]$, o bien en top, lo que nos lleva a la recurrencia $T(u) = T(\sqrt{u}) + O(1) = O(\log \log u)$.

4.2.3. Inserciones

Para simplificar, supongamos que sabemos que el elemento y que queremos insertar no está ya en X. Descomponemos $y = a\sqrt{u} + b$, y entonces debemos insertar b en $bottom[a]$, recursivamente. Sin embargo, si $bottom[a]$ estaba vacío (es decir, $bottom[a].size = 0$ antes de la inserción), también debemos insertar a en top, recursivamente. Finalmente, incrementamos $size$ y actualizamos los campos min y max para considerar el nuevo elemento y (min y max son válidos sólo si $size > 0$).

Si bien este método de inserción es correcto, tiene el problema de que podemos realizar dos invocaciones recursivas sobre subuniversos de tamaño $\sqrt{u}$: en $bottom[a]$ y también en top. Esto nos lleva a la recurrencia $T(u) = 2\,T(\sqrt{u}) + O(1) = O(\log u)$, es decir, el costo de inserción se hace demasiado alto.

Para evitar este problema basta un pequeño retoque a la estructura: el valor max dejará de ser *una copia* del máximo valor almacenado. Ahora será un valor que se mantiene aparte y no se almacena dentro del vEB.

Con este nuevo invariante, la inserción procede de la siguiente forma. Primero, si $size = 0$, todo lo que hacemos es $size \leftarrow 1$, $min \leftarrow y$, y $max \leftarrow y$. Como el elemento y se guardó en max, no lo almacenamos en su subuniverso correspondiente. Si, en cambio, la estructura tiene elementos, lo primero que hacemos es ver si y no es el nuevo máximo: si $y > max$, intercambiamos sus valores. De esta forma, el elemento insertado pasa a estar guardado en max, y el que teníamos en max (que ya no es más el máximo) pasa a ser el que tenemos que insertar en el árbol. Luego realizamos la inserción de y tal como la habíamos descrito inicialmente (salvo que las variables que se actualizan al final son sólo min y $size$).

Con este nuevo método, todavía podemos realizar dos llamadas recursivas, pero si eso ocurre, una de ellas toma tiempo $O(1)$: si insertamos a en top, es porque $bottom[a].size = 0$ cuando le insertamos b, y con nuestro nuevo algoritmo, la inserción de b en $bottom[a]$ toma tiempo $O(1)$. De este modo, el tiempo de la inserción se hace $O(\log \log u)$.

El nuevo invariante requiere un pequeño cambio en la búsqueda del predecesor de y: antes que nada, debemos verificar si $size > 0$ y $y \geq max$, en cuyo caso el predecesor de y es max.

De otro modo, procedemos como antes.

4.2.4. Borrados

Para simplificar, supondremos que el elemento y que queremos borrar está en la estructura. Lo primero que haremos será verificar si $size = 1$, en cuyo caso sabemos que el elemento a borrar es el único almacenado, max. En este caso basta que hagamos $size \leftarrow 0$ para invalidarlo.

Si tenemos más de un elemento, la siguiente verificación es si no estamos borrando el máximo, $y = max$. En este caso, debemos encontrar el nuevo máximo del conjunto, que no es difícil de calcular: $max \leftarrow top.max\sqrt{u} + bottom[top.max].max$. Ahora este elemento debe eliminarse del árbol, por lo que asignamos $y \leftarrow max$ antes de continuar. De otro modo, mantenemos el elemento original y a borrar.

Para borrar y del árbol, lo descomponemos en $y = a\sqrt{u} + b$, y borramos b de $bottom[a]$, recursivamente. Si luego de este borrado tenemos que $bottom[a].size = 0$, entonces debemos también borrar a de top, recursivamente.

Finalmente, decrementamos $size$ y recalculamos el nuevo mínimo, que puede haber cambiado si $y = min$. En este caso, actualizamos $min \leftarrow top.min\sqrt{u} + bottom[top.min].min$.

Nuevamente, el borrado en un universo de tamaño u puede realizar dos llamadas recursivas en subuniversos de tamaño $\sqrt{u}$, pero al igual que en la inserción, sólo una de ellas puede ser no trivial: si borramos a de top, es porque $bottom[a]$ se hizo vacío luego de borrar, por lo cual el borrado de b en $bottom[a]$ se hizo en tiempo $O(1)$.

4.3. Diccionarios de Strings

Cualquier implementación de diccionarios, por ejemplo, un árbol balanceado o una tabla hash, pueden usarse para almacenar un conjunto de strings. El primero ofrecería tiempo $O(m \log n)$ de peor caso y el segundo $O(m)$ promedio para insertar, borrar, o buscar un string de largo m en un conjunto de n strings. Aquí mostraremos cómo aprovechar el hecho de que los objetos son strings para operar en tiempo de peor caso $O(m)$, pudiendo además buscar todos los strings prefijados por un string de consulta. Luego usaremos la estructura de datos para indexar un texto, de modo de poder buscar en sus substrings.

4.3.1. Tries

Un *trie*, *árbol digital*, almacena un conjunto de strings. En el trie, cada nodo puede tener cero o más hijos. La arista hacia cada hijo está rotulada por un carácter del alfabeto, y no puede haber dos aristas saliendo de un mismo nodo y rotuladas por el mismo carácter. Cada hoja almacena uno de los strings del conjunto. Los strings almacenados se suponen terminados por un carácter especial "$", lo que impide que un string sea prefijo de otro.

Para cada nodo v, llamaremos $str(v)$ al string que se obtiene concatenando los rótulos de las aristas en el camino de la raíz hasta v. Así, si v es la raíz, $str(v)$ es la cadena vacía, y si v es una hoja que representa el string S, $str(v) = S$. Para los nodos internos v, $str(v)$ es un prefijo común que comparten todos los strings almacenados en sus hojas descendientes (note que dos strings que compartan un prefijo común de largo ℓ deben compartir los primeros ℓ nodos en el camino desde la raíz hasta sus hojas).

Para buscar una determinada cadena $S[1..m]$ en un trie, buscaremos en realidad $S^* = S\$$. Partimos de la raíz v_0 y bajamos por la arista rotulada $S^*[1]$ para llegar al nodo v_1. De v_1 bajamos por la arista rotulada $S^*[2]$ para llegar al nodo v_2, y así. Esta búsqueda termina de dos formas posibles:

1. Un cierto nodo v_i del camino no tiene un hijo rotulado $S^*[i+1]$, para $0 \leq i \leq m$. Esto significa que S no está en el conjunto.

2. Llegamos al nodo v_{m+1}, que debe ser una hoja pues desciende del símbolo especial \$. En este caso, S está en el conjunto, pues $str(v_{m+1}) = S^*$.

La búsqueda de S requiere, entonces, tiempo $O(m)$. Esto supone que podemos encontrar en tiempo constante la arista rotulada $S[i]$. Si el tamaño σ del alfabeto se considera constante, entonces esto es inmediato con cualquier implementación que usemos para los hijos de los nodos. Si no, aún podemos conseguir tiempo constante si, por ejemplo, cada nodo guarda un arreglo de sus σ hijos, lo cual sin embargo requiere espacio $O(\sigma)$ por nodo. Si el trie no sufre modificaciones, podemos usar hashing perfecto (que veremos más adelante) para mantener el tiempo constante y usar espacio proporcional al número de hijos de cada nodo (cobrándole este espacio a los hijos, esto amortiza a $O(1)$ por nodo en total). Si el trie sufre modificaciones, aún podemos usar hashing (no perfecto) para tener tiempo $O(1)$ en promedio. Finalmente, podemos almacenar los hijos de cada nodo en un árbol balanceado para bajar al hijo en tiempo $O(\log \sigma)$, de modo que el tiempo total de búsqueda sea $O(m \log \sigma)$. Por simplicidad, ignoraremos este tiempo de ahora en adelante.

Reduciendo el tamaño. Si guardamos n strings de largo total N, el trie podría llegar a tener N nodos. Esto hace del trie una estructura bastante exigente en espacio. Una forma de reducirlo significativamente es eliminar los caminos de nodos con un solo hijo (llamémoslos *unarios*) que terminan en una hoja, es decir, si $v_1 \to v_2 \to v_3 \ldots \to v_t$ es un camino donde v_t es una hoja y $v_2, \ldots, v_{t-1}$ tienen un solo hijo, entonces los eliminamos y ponemos a v_t como hijo directo de v_1 (conservando el rótulo de la arista $v_1 \to v_2$). A cambio, deberemos almacenar el string en la hoja v_t (lo que en muchos casos se hace de todos modos). Con esta modificación, la búsqueda de S en el trie puede nuevamente terminar de dos formas posibles:

1. Un cierto nodo interno v_i del camino no tiene un hijo rotulado $S^*[i+1]$, para $0 \leq i \leq m$. Igual que antes, esto significa que S no está en el conjunto.

2. Un cierto nodo v_i del camino es una hoja. Esto significa que $str(v_i)$ es el único string del conjunto prefijado por $S^*[1..i]$. Ahora debemos comparar $S^*[i+1..m+1]$ con $str(v_i)[i+1..m+1]$, para determinar si son iguales o no.

Nuevamente, el tiempo de búsqueda es $O(m)$. Con esta modificación, un trie todavía podría tener $\Theta(N)$ nodos en el peor caso, pero en promedio tendrá sólo $O(n)$ nodos si se insertan strings aleatorios.

Búsqueda de prefijos. Otra operación que permite el trie es la *búsqueda de prefijos*, es decir, contar o listar todos los strings del conjunto prefijados por S. En este caso debemos descender usando S, no S^*. Si la búsqueda termina en el caso 1, no hay strings en el conjunto prefijados por S. Si termina en el caso 2, y $S[i+1..m] = str(v_i)[i+1..m]$, entonces $str(v_i)$ es el único string del conjunto prefijado por S. Pero, al no estar S terminado con \$, existe una tercera posibilidad: que la búsqueda termine en un nodo interno v_m después de haberse consumido todos los caracteres de $S[1..m]$. En ese caso, toda hoja que descienda de v_m está prefijada por S. Podemos hacer que cada nodo almacene el número de hojas que descienden de él, resolviendo esta búsqueda en tiempo $O(m)$. Si en cambio queremos poder listar esos strings, entonces cada nodo debería almacenar un puntero a su primera y última hoja descendiente, y cada hoja un puntero a su siguiente hoja. Con ello, cada string prefijado por S se alcanza en tiempo constante una vez realizada la búsqueda. Los detalles de cómo mantener estos punteros los dejaremos como ejercicio.

Inserciones y borrados. Veremos ahora cómo insertar y borrar en esta estructura. La inserción de un string $S[1..m]$ parte como una búsqueda de S^*, y termina de una de las dos formas vistas. Veamos qué hacer en cada caso:

1. Si el nodo interno v_i no tiene un hijo rotulado $S^*[i+1]$, entonces agregamos ese hijo, el cual será una hoja conteniendo el string S^*.

2. Si llegamos a una hoja v_i conteniendo $S'[1..m'+1]$ (también terminada con \$), comparamos $S^*[i+1..m+1]$ con $S'[i+1..m'+1]$ hasta encontrar la primera diferencia $S^*[j] \neq S'[j]$ (si no hay diferencia, entonces S ya estaba en el conjunto). Creamos entonces una cadena de hijos $v_i \to v_{i+1} \to \ldots \to v_{j-1}$, donde cada arista $v_k \to v_{k+1}$ está rotulada $S^*[k+1]$ $(= S'[k+1])$. De v_{j-1} salen dos hijos: $v_{j-1} \to v_j$, rotulada $S^*[j]$, lleva a la hoja v_j que contiene el string S^*, mientras que $v_{j-1} \to v'_j$, rotulada $S'[j]$, lleva a la hoja v'_j que contiene el string S'.

En todos los casos, el tiempo total sigue siendo $O(m)$. Un caso especial es cuando el trie no contiene strings, en cuyo caso la primera inserción hace que el trie consista de una única hoja que almacena el string.

La eliminación del string S parte como una búsqueda exitosa de S^* , llegando a la hoja v que lo contiene y eliminándola. Luego, debemos asegurarnos de que no nos quede un camino

unario hacia una hoja. Si v tiene un único hermano v' y v' es una hoja, entonces entramos en el siguiente ciclo: Mientras el padre u de v' tenga un solo hijo, eliminamos u y colgamos a v' del padre de u. Si llegamos a eliminar la raíz, entonces el trie consiste de una única hoja, v'. Note que el tiempo de borrado sigue siendo $O(m)$, incluso con el ciclo que recorta los caminos unarios.

4.3.2. Árboles Patricia

Si bien el peor caso de espacio $O(N)$ en un trie es bastante raro cuando eliminamos los caminos unarios hacia las hojas, puede ocurrir con algunos conjuntos de strings que comparten prefijos largos (por ejemplo, conjuntos de URLs). El *árbol Patricia* (también conocido como *blind trie*) es una variante del trie que asegura espacio $O(n)$ independientemente del largo de los strings, y mantiene el tiempo de las operaciones en $O(m)$. El precio es que se deben almacenar los strings completos en alguna parte (obviamente).

El invariante del árbol Patricia es que *no hay caminos unarios* de ninguna clase (tampoco entre nodos internos). Como todo nodo tiene al menos dos hijos y hay n nodos hojas, hay menos de n nodos internos, con lo cual el total de nodos es $O(n)$.

Para ello, cada arista estará rotulada en general con un string, no con un solo carácter, pues potencialmente puede provenir de la eliminación de un camino unario. El árbol Patricia no almacena el string completo, sino sólo su primera letra a y su largo ℓ, en un par (a, ℓ). Esto hace que no haya suficiente información en el trie para que la búsqueda de S^* compare el string completo, sino solamente algunos caracteres: si hemos comparado $S^*[1..i]$ y la arista rotulada por $S^*[i+1]$ es de la forma $(S^*[i+1], \ell)$, entonces deberemos seguirla y en su hijo suponer que hemos consumido ya $S^*[1..i+\ell]$. Esto hace que, al llegar a una hoja v, debamos comparar completamente S^* con $str(v)$, pues algunos de los caracteres que "saltamos" podrían no haber coincidido. Lo que sabemos es que, si S está presente, S^* tiene que estar en v, pues no hay otra hoja que coincida con S^* en los caracteres que sí vimos.

Para buscar S en un árbol Patricia, bajaremos por los nodos $v_0, v_1, \ldots$ buscando S^*, de la forma que acabamos de explicar (es decir, saltando los largos ℓ indicados en las aristas), hasta que ocurra una de las siguientes situaciones:

1. Un cierto nodo interno v_i del camino no tiene un hijo rotulado $S^*[j]$, donde $1 \leq j \leq m+1$ es el carácter que toca comparar en v_i. Igual que antes, esto significa que ni S ni su prefijo $S[1..j]$ están en el conjunto.

2. Llegamos a un nodo interno v_i donde el carácter que toca comparar es $j > m+1$. Esto significa que todos los strings que descienden de v_i son más largos que S^*, y por lo tanto S no está en el conjunto.

3. Llegamos a una hoja v_k. Si el siguiente carácter a consumir en S^* luego de procesar la arista es $j \neq m+2$, entonces la hoja no contiene S^* sino un string más corto o más largo, en cuyo caso ya sabemos que S no está en el conjunto. En caso de pasar este

test, debemos comparar *la totalidad* de S^* con $str(v_k)$, por si algún carácter saltado no coincide. Estos dos strings son iguales sii S está en el conjunto.

A pesar de la doble verificación, el tiempo de búsqueda sigue siendo $O(m)$.

Búsqueda de prefijos. Para buscar prefijos, usamos $S[1..m]$ en vez de S^*. En el caso 3 de arriba, el único string posible prefijado por S es $str(v_k)$. Para ello, la posición del siguiente carácter a comparar de S debe ser $> m+1$, pues si no $str(v_k)$ es más corto que S. Pasado este test, debemos comparar la totalidad de S con $str(v_k)[1..m]$ para determinar si S es prefijo de $str(v_k)$, dado que podemos no haber comparado todos los caracteres de S.

Debemos también modificar el caso 2, deteniendo la búsqueda cuando el carácter a comparar es el $j > m$. Para resolver este caso, cada nodo interno v_i debe apuntar a alguna hoja v_k que descienda de él, en un campo $hoja(v_i)$ (supondremos que, si v_k es una hoja, entonces $hoja(v_k) = v_k$ sin necesidad de almacenarla). Calcularemos entonces $v_k = hoja(v_i)$ y verificaremos si $S = str(v_k)[1..m]$. De ser así, S es prefijo de *todos* los strings almacenados en hojas que descienden de v_i, y si no, de *ninguno*. Nuevamente, podemos almacenar contadores en cada v_i para entregar la cantidad de strings prefijados en tiempo $O(m)$. No necesitamos, en cambio, punteros para recorrer las hojas eficientemente, pues si hay h hojas descendiendo de v_i, su subárbol contiene $O(h)$ nodos, por lo cual podemos simplemente recorrerlo recursivamente.

Inserciones y borrados. La inserción en un árbol Patricia es un proceso más complejo que en un trie. Primero, debemos buscar S^* en el árbol, y no encontrarlo. Esto ocurre según los casos 1 a 3 de la búsqueda:

1. Un cierto nodo interno v_i del camino no tiene un hijo rotulado $S^*[j]$, donde $1 \leq j \leq m+1$ es el carácter que toca comparar en v_i. En este caso, debemos tomar $v_k = hoja(v_i)$ y comparar S^* con $str(v_k)$, para determinar la primera posición $d \leq j$ donde S^* difiere de todos los strings en las hojas que descienden de v_i.

2. Llegamos a un nodo v_i donde ya nos pasamos de la posición $m+1$. Procedemos exactamente como en el caso 1.

3. Llegamos a una hoja v_k. Nuevamente, comparamos S^* con $str(v_k)$ para determinar d.

Note que el punto de inserción de S^* corresponde a $S^*[d]$, el cual puede estar más arriba del nodo v_i o v_k, pues llegamos a esos nodos suponiendo que los caracteres no revisados coincidían. Una vez conocido d, pueden pasar tres cosas:

1. Estábamos en un nodo interno v_i y resulta que $j = d$. En este caso, creamos un nuevo hijo hoja de v_i asociado al string S^*, y rotulamos la arista que los une con $(S[d], m+2-d)$.

2. Estábamos en una hoja v_k, a la que llegamos luego de examinar el carácter $S[d']$, y resulta $d > d'$. En ese caso, convertimos a v_k en padre de dos hojas: una representa el string original S' asociado a v_k y la otra representa S^*. La arista hacia la primera se rotula $(S'[d], |S'| - d + 1)$, y la arista hacia la segunda se rotula $(S^*[d], m + 2 - d)$. Finalmente, la arista $(a, |S'| - d' + 1)$ que llegaba a v_k debe ser convertida a $(a, d - d')$ y $hoja(v_k)$ puede ser asignada a cualquiera de las dos hojas creadas.

3. Ninguna de las anteriores, en cuyo caso debemos retroceder en la recursión hasta llegar al ancestro v_r desde el que pasamos a v_s por una arista rotulada (a, ℓ). Al pasar de v_r a v_s, pasamos de haber consumido $S^*[1..d']$ a haber consumido $S^*[1..d' + \ell]$, con $d' < d - 1 < d' + \ell$. Entonces debemos cortar esta arista en dos, introduciendo un nuevo nodo v hijo de v_r y padre de v_s. La arista de v_r a v se rotula $(a, d - 1 - d')$, y la de v a v_s se rotula $(str(v_k)[d], d' + \ell - d + 1)$, donde $v_k = hoja(v_s)$. Se copia $hoja(v) \leftarrow hoja(v_s)$. Finalmente, se crea una nueva hoja v' que también es hija de v, con la arista entre ellas rotulada $(S^*[d], m + 2 - d)$, y se asocia la hoja a S^*.

Para borrar S, una vez eliminada la hoja v que representaba S^*, tenemos que verificar si su padre no queda con un solo hijo, y de ser así eliminar ese padre, reemplazando al padre de ese otro hijo hoja por su abuelo. El proceso no necesita repetirse hacia arriba. Concretamente, si u es el padre de v y u tiene sólo otro hijo v', por una arista rotulada (a, ℓ), entonces sea u' el padre de u, conectados por una arista rotulada (a', ℓ'). Entonces se elimina u y se cuelga a v' de u', por una arista rotulada $(a', \ell + \ell')$.

Asimismo, debemos eliminar referencias de la forma $hoja(v_i) = v$ en cualquier ancestro v_i de v. Para ello, antes de posiblemente eliminar al padre u de v, tomamos $w = hoja(v')$ para cualquier hijo $v' \neq v$ de u. Luego de posiblemente eliminar al padre de v, partimos del (tal vez nuevo) padre de v' y recorremos el camino hasta la raíz, reemplazando cualquier $hoja(v_i) = v$ por $hoja(v_i) \leftarrow w$.

En todos los casos, la inserción y el borrado de un string $S[1..m]$ cuesta $O(m)$.

Alfabeto binario. Una implementación particularmente simple de los árboles Patricia se obtiene si consideramos el alfabeto como binario, es decir, cada carácter en el alfabeto $[1..\sigma]$ se toma como una cadena de $\log_2 \sigma$ bits. Esto hace que cada nodo interno del árbol tenga exactamente 2 hijos, simplificando la implementación. Note que el espacio sigue siendo $O(n)$.

El precio es que el camino de la raíz a la hoja donde se almacena $S[1..m]$ puede llegar a tener largo $O(m \log \sigma)$, que será ahora la nueva complejidad de todas las operaciones. Como vimos al comienzo, esta es realmente la complejidad en algunas implementaciones.

Autocompletado. Una aplicación interesante de árboles Patricia es el autocompletado automático, que a medida que uno tipea una palabra va proponiendo la palabra más probable que uno querría terminar de escribir. El usuario puede, en cualquier momento, aceptar la opción que le ofrece el autocompletado o seguir tipeando. Y, obviamente, puede terminar

tipeando algo que no esté aún en el conjunto de palabras conocidas. Almacenando las palabras conocidas y sus frecuencias en un árbol Patricia se puede implementar el autocompletado en forma bastante simple.

Al comenzar a escribirse una nueva palabra, partimos de la raíz del árbol y vamos bajando a medida que se tipean nuevos caracteres. Si estamos en un nodo v, podemos proponer como autocompletado la hoja que lleva mayor frecuencia acumulada y que desciende de v. Para ello, necesitamos que v tenga un campo $best(v)$ que apunta a esa hoja. Si no se acepta esta sugerencia y en cambio se tipea una a, buscaremos la arista de la forma (a, ℓ) que salga de v hacia u, y propondremos $best(u)$. El proceso termina de dos formas posibles:

1. Se llega a una hoja w, o bien porque en algún punto se acepta el autocompletado desde un nodo v con $best(v) = w$, o bien porque se tipea hasta llegar a w. Las hojas deben tener un campo $freq(w)$ que indique la cantidad de veces que han aparecido. Debemos entonces incrementar $freq(w)$.

2. Nunca se acepta el autocompletado y se termina tipeando una palabra nueva, que no estaba en el diccionario. Se debe entonces crear la hoja correspondiente w, con $freq(w) = 1$.

Luego de haber visitado o creado una hoja w, debemos actualizar los campos $best(v)$ de sus ancestros. Para ello, volvemos desde la hoja w hasta la raíz asignando, para cada ancestro v, $best(v) \leftarrow w$ si es que $freq(best(v)) \leq freq(w)$ (notar que, ante la igualdad, preferimos la palabra tipeada más recientemente; podemos usar otras políticas también). Este proceso se puede detener apenas encontremos un v que no reemplace su $best(v)$, pues sus ancestros tampoco lo harán. Como puede que hayamos llegado a w directamente desde un ancestro v con $best(v) = w$, este recorrido hacia arriba desde w requiere o bien punteros hacia el padre de cada nodo, o bien que volvamos a entrar al árbol con el string $str(w)$.

4.3.3. Árboles de sufijos

Consideremos un texto $T[1..n]$ terminado con el carácter especial \$. Este texto define n *sufijos* $T[i..n]$, para $1 \leq i \leq n$. Si insertamos todos estos sufijos en un árbol Patricia, el resultado es el *árbol de sufijos* de T. La única diferencia es que las hojas del árbol, en vez de almacenar el sufijo $T[i..n]$, simplemente almacenan la posición i del texto, con lo cual se puede acceder directamente al sufijo.

Si hacemos una búsqueda de prefijo por $S[1..m]$, obtendremos la cantidad de sufijos de T prefijados por S, en tiempo $O(m)$. Eso es exactamente lo mismo que la cantidad de ocurrencias de S en T. Asimismo, podemos listar las posiciones iniciales i de los sufijos $T[i..n]$ prefijados por S, que es lo mismo que las posiciones iniciales de las ocurrencias de S en T. Cada una de esas posiciones se lista en tiempo $O(1)$, tal como vimos.

En resumen, el árbol de sufijos es una estructura de datos que, construida sobre un texto de largo n, ocupa espacio $O(n)$ y permite buscar las ocurrencias de substrings en el texto en

tiempo óptimo. El árbol de sufijos se puede construir mediante insertar los sufijos de T uno a uno. Sobre un texto aleatorio, la altura del árbol es $O(\log_\sigma n)$, por lo cual la construcción toma tiempo esperado $O(n \log n)$. En el peor caso, sin embargo, esta construcción puede requerir tiempo $O(n^2)$ (por ejemplo, un texto T con substrings repetidos muy largos). Existen varios algoritmos de construcción del árbol de sufijos en tiempo $O(n)$; daremos algunas referencias al final del capítulo.

Además de buscar ocurrencias de strings en T, el árbol de sufijos permite responder muchas otras preguntas más complejas. Describiremos una como ejemplo.

Autorrepeticiones relevantes. Para determinación de plagio y autoría, detección de genes comunes entre especies, etc. un tipo de pregunta de importancia es qué cadenas se repiten significativamente en un texto T (que puede ser la concatenación de varias secuencias).

La autorrepetición más larga de T, es decir, la cadena más larga que aparece dos veces en T, se puede encontrar fácilmente con un recorrido en profundidad por el árbol de sufijos. Extendamos el concepto de $str(v)$ a nodos internos también, de modo que $str(v)$ es la concatenación de los strings que rotulan el camino de la raíz hasta v. Cada nodo interno v del árbol se corresponde con un string distinto $str(v)$ que aparece al menos dos veces (pues si no, v no tendría al menos dos hijos y no sería un nodo interno). Lo que queremos, entonces, es el nodo interno v que maximice $|str(v)|$. Este largo se puede calcular fácilmente mediante ir acumulando los campos ℓ de los rótulos (a, ℓ) a medida que bajamos en el árbol. En total, en tiempo $O(n)$ recorremos todo el árbol y determinamos un nodo v que maximiza $|str(v)|$. El string mismo se puede recuperar mediante ir a $hoja(v)$ y rescatar la posición i donde empieza el sufijo, de modo que el string repetido más largo es $T[i..i + |str(v)| - 1]$.

De la misma manera, si almacenamos un campo $size(v)$ con el número de hojas que tiene el nodo v, podemos por ejemplo encontrar todos los strings de largo $\geq \ell$ que aparecen al menos m veces en T, para cualquier parámetro ℓ y m.

4.3.4. Arreglos de sufijos

Si bien el árbol de sufijos ocupa espacio $O(n)$, la constante es bastante grande: puede haber hasta n nodos internos (cada uno con hasta 6 campos enteros) y n hojas (cada una con 2 campos). Si un entero ocupa 4 bytes, el espacio suma hasta $32n$ bytes, mientras que T usa típicamente n bytes.

El *arreglo de sufijos* es una estructura que conserva algo de la funcionalidad del árbol de sufijos, pero sólo requiere un entero por posición de T. La estructura es un arreglo $A[1..n]$ de enteros, donde $A[i]$ apunta al i-ésimo sufijo en orden lexicográfico. Es decir, A es una permutación de $[1..n]$ donde $T[A[i]..n] < T[A[i + 1]..n]$ para todo $1 \leq i < n$.

Los sufijos que empiezan con un cierto string S forman un rango lexicográfico, por lo cual aparecen en un rango de A. Para encontrar las ocurrencias de $S[1..m]$ en T podemos entonces hacer dos búsquedas binarias para encontrar los dos extremos del rango $A[sp..ep]$ de los sufijos que empiezan con S. Una vez encontrado el rango $A[sp..ep]$, sabemos que S

ocurre $ep - sp + 1$ veces en T, y que las posiciones iniciales de las ocurrencias son $A[sp]$, $A[sp+1]$, ..., $A[ep]$.

Cada paso en la búsqueda binaria en un cierto $A[i]$ requiere comparar S con el string $T[A[i]..A[i]+m-1]$, para determinar si continuar a la derecha o a la izquierda de i. Como estas comparaciones requieren examinar hasta m caracteres, el tiempo de encontrar este rango es $O(m \log n)$. Esta mayor complejidad es uno de los precios de haber reducido el espacio del árbol de sufijos a sólo n enteros. Otro precio es que no podemos realizar ciertas búsquedas complejas en forma tan eficiente, como la que describimos para encontrar las repeticiones relevantes de T.

El arreglo de sufijos se puede construir con cualquier algoritmo de ordenamiento. Cada una de las $O(n \log n)$ comparaciones de strings requiere comparar $O(\log_\sigma n)$ caracteres en un texto promedio (donde cada carácter se genera uniformemente), con lo cual el costo promedio total es $O(n \log n \log_\sigma n)$. En un texto con muchos substrings repetidos, sin embargo, esto puede empeorar hasta $O(n^2 \log n)$. Existen algoritmos especializados que construyen el arreglo de sufijos en tiempo $O(n)$; damos algunas referencias al final del capítulo.

4.4. Ficha Resumen

Con n elementos en un universo entero $[1..u]$, o n strings de largo total $N \geq n$, sobre un alfabeto constante, insertando/borrando/buscando un string de largo m:

- Counting y bucket sort: $O(n+u)$.
- Radix sort: $O(n \log_n u)$.
- Ordenando strings: $O(N)$.
- Van Emde Boas trees: espacio $O(n + \frac{u}{\log^c u})$ y tiempo $O(c \log\log u)$ para cualquier constante c, para insertar, borrar, y buscar predecesor. Espacio $O(n)$ para sólo buscar en tiempo $O(\log\log \frac{u}{n})$, o para las tres operaciones en tiempo esperado $O(\log\log u)$.
- Tries: espacio $O(N)$ en el peor caso y tiempo $O(m)$ para insertar, borrar y buscar.
- Árboles Patricia: espacio $O(n)$ y tiempo $O(m)$ para insertar, borrar y buscar.
- Árboles de sufijos: espacio $O(n)$ para un texto de largo n y buscando en tiempo $O(m)$.
- Arreglos de sufijos: espacio $O(n)$ para un texto de largo n (pero menor constante que el árbol de sufijos) y buscando en tiempo $O(m \log n)$.

4.5. Material Suplementario

Cormen et al. [CLRS01, sec. 8.2 a 8.4] presentan el bucket sort (al que llaman counting sort) y el radix sort. Luego presentan, con el nombre de bucket sort, una estrategia que ordena en tiempo $O(n)$ promedio si los valores se distribuyen uniformemente (y puede ordenar números reales, no solamente en universos discretos). Mehlhorn y Sanders [MS08, sec. 5.6] presentan, con menos detalle, el bucket sort (que llaman Ksort), el radix sort, y el sort de $O(n)$ promedio (que llaman uniform sort). Aho et al. [AHU83, sec. 8.5] también presentan el bucket sort (al que llaman bin sort) y el radix sort, y mencionan la aplicación a ordenar strings del mismo largo. Sedgewick [Sed92, cap. 8] presenta el bucket sort, al que llama distribution counting, y luego [Sed92, cap. 10] el radix sort, al que llama straight radix sort para diferenciarlo del que llama radix exchange sort, que se parece más a un QuickSort que va particionando por el bit más alto, luego por el segundo más alto, etc. Manber [Man89, sec. 6.4.1] describe brevemente el bucket sort (¡con ese nombre!) y las mismas variantes de radix sort que Sedgewick. Aho et al. [AHU74, sec. 3.2] presentan el bucket sort (con ese nombre) y luego el radix sort, al que llaman lexicographic sort porque se concentran en strings, primero de largo fijo y después de largo variable (presentando un algoritmo muy similar al que vimos). Varios autores [CLRS01, MS08] mencionan la posibilidad de realizar radix sort partiendo por el símbolo más significativo primero (MSD radix sort), particionando así el arreglo y luego ordenando recursivamente cada partición, pero advierten que esto genera muchas particiones muy pequeñas, lo que hace que el método que vimos (LSD radix sort) sea preferible. Note que el MSD radix sort es conceptualmente similar a construir un trie sobre las secuencias de símbolos. Por último, sólo algunos autores [CLRS01, Sed92] muestran cómo el bucket sort puede convertir los contadores en punteros para escribir el output ordenado en un arreglo; los otros usan estructuras más ineficientes como colas o listas enlazadas para los buckets.

Cormen et al. [CLRS01, Cap. 20 de la tercera edición] describen los árboles de van Emde Boas. También se describen en varias fuentes online que listamos al final.

Aho et al. [AHU83, sec. 5.3] hacen una presentación muy básica y detallada de los tries. Sedgewick [Sed92, cap. 17] entrega un tratamiento más completo, que incluye árboles Patricia. Sedgewick habla de tiempos de operación $O(\log n)$ para el árbol Patricia, basándose en que la profundidad promedio de los nodos cuando se insertan strings aleatorios es $O(\log_\sigma n)$. Aho et al. [AHU74, sec. 9.5] presentan los position trees, que son esencialmente tries con todos los sufijos de un texto, con aplicaciones y algoritmo de construcción. Para árboles y arreglos de sufijos es necesario consultar libros de stringología. Crochemore y Rytter [CR02, cap. 4 y 5] presentan el árbol de sufijos, algoritmos de construcción y algunas aplicaciones, así como otras estructuras relacionadas llamadas autómatas de sufijos o DAWGs (directed acyclic word graphs) [CR02, cap. 6]. Estas estructuras hacen evidentes las conexiones entre los tries y los autómatas finitos, así como las estructuras que usan los algoritmos de búsqueda en texto como Knuth-Morris-Pratt y Aho-Corasick. Crochemore et al. [CHL07, cap. 5] también cubren tries, árboles y autómatas de sufijos. También dedican un capítulo [CHL07,

cap. 4] al arreglo de sufijos, su construcción en tiempo lineal, y varias extensiones. Otros capítulos de ambos libros [CR02, CHL07] exploran varias aplicaciones de estas estructuras.

Otras fuentes online de interés:

- brilliant.org/wiki/radix-sort
- www.bowdoin.edu/~ltoma/teaching/cs231/duke_cps130/Lectures/L07.pdf
- www.youtube.com/watch?v=Nz1KZXbghj8
- www.ics.uci.edu/~eppstein/161/960123.html
- ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/lecture-notes/MIT6_046JS15_lec04.pdf
- www.cs.bris.ac.uk/~bs4039/slidesAA/aa-12.pdf
- www2.hawaii.edu/~nodari/teaching/s16/notes/notes10.pdf
- www.youtube.com/watch?v=hmReJCupbNU
- www.youtube.com/watch?v=ZrV7GiuMNo4
- web.stanford.edu/class/cs166/lectures/03/Small03.pdf
- algs4.cs.princeton.edu/lectures/52Tries.pdf
- www.cs.cmu.edu/~avrim/451f07/lectures/lect1002.pdf
- www.youtube.com/watch?v=NinWEPPrkDQ
- visualgo.net/en/suffixtree y visualgo.net/en/suffixarray

Capítulo 5

Algoritmos en Línea

Hay casos en que un algoritmo debe tomar decisiones antes de conocer todo el input. Un ejemplo de la vida real es cuando estamos esperando el bus sin saber cuándo llegará: ¿nos conviene seguir esperando o irnos caminando? Estos algoritmos que toman decisiones teniendo parte del input se llaman *algoritmos en línea*, y la forma de analizarlos es medir su *competitividad* en comparación con un algoritmo *óptimo* que conoce todo el input de antemano. Note que no estamos hablando de la eficiencia en tiempo o espacio de los algoritmos, sino de la calidad del resultado que producen.

Tomemos el ejemplo del bus. El bus llega a destino en tiempo x y caminando llego en tiempo $y \gg x$. Pero el bus tardará tiempo z en llegar a la parada, y yo no cononzco z. Un algoritmo óptimo, que conociera z, simplemente elegiría la mejor alternativa, tardando mín$(x+z, y)$. ¿Qué alternativas tengo si no conozco z? Puedo irme caminando inmediatamente y tardar y, en cuyo caso podría ser que $z = 0$ y el algoritmo óptimo tarda x, es decir, es $\frac{y}{x}$ veces más rápido. Puedo esperar el bus todo lo que sea necesario, tardando $x + z$, pero entonces si z es muy grande el algoritmo óptimo se irá caminando directamente, tardando y y siendo $\frac{x+z}{y}$ veces más rápido. ¿Puedo tener una estrategia intermedia? Sí: puedo esperar y, y si no llegó el bus, me voy caminando. Esto demora $x + z$ si $z < y$, y $2y$ si no. Note que esto nunca es más que el doble de lo que demoraría la estrategia óptima: si el óptimo es $x + z < y$, nosotros también nos iremos en bus porque $z < y$; si el óptimo es $y < x + z$, nosotros tardamos o bien $2y$, o si $z < y$, $x + z < 2y$. Decimos que nuestra solución es un algoritmo en línea 2-competitivo.

Dado un algoritmo en línea A y un algoritmo óptimo OPT, que producen soluciones de costo $C_A(I)$ y $C_{OPT}(I)$ para un input I, diremos que A *es c-competitivo* si existe una constante b tal que, para todo posible input I, se cumple que

$$C_A(I) \quad \leq \quad c \cdot C_{OPT}(I) + b.$$

Note que la competitividad así definida es una medida de peor caso. Existen medidas similares para algoritmos aleatorizados, pero no las veremos en el apunte.

5.1. Aplicaciones Simples

5.1.1. Subastas y codificación de enteros

Considere un juego tipo subasta donde un artículo tiene un precio oculto x. Los participantes hacen ofertas y. Si $y < x$, la casa se queda con y pero no entrega nada a cambio. Si $y \geq x$, la casa se queda con y y entrega el artículo. Un participante puede ofertar varias veces hasta obtener el artículo. Obviamente el algoritmo óptimo, que conoce x, oferta x y se lleva el artículo. ¿Cuál es la mejor estrategia para un algoritmo en línea?

Consideremos una estrategia de tipo exponencial, en que en el turno k ofertamos c^k, para $k = 0, 1, 2, \ldots$. Lo peor que puede pasar es que x sea de la forma $c^n + 1$, de modo que al ofertar c^n lo perdamos, y debamos ofertar c^{n+1}. En ese caso el costo total en que incurrimos fue

$$\sum_{k=0}^{n+1} c^k \quad = \quad \frac{c^{n+2} - 1}{c - 1},$$

mientras que el algoritmo óptimo gasta $c^n + 1$. Para simplificar las cuentas podemos reemplazar pesimistamente el costo del algoritmo en línea por $\frac{c^{n+2}}{c-1}$ y el del óptimo por c^n. El cociente entre ambos es entonces

$$\frac{c^{n+2}}{(c-1)c^n} \quad = \quad \frac{c^2}{c-1},$$

que se minimiza con $c = 2$. Es decir, ofertando potencias de 2 tenemos un algoritmo 4-competitivo. No es difícil ver que un algoritmo que vaya ofertando subexponencialmente o superexponencialmente no lograrán una competitividad constante, por lo que esta es la mejor competitividad que podemos conseguir.

Codificación γ. Consideremos ahora un juego análogo, en el que se tiene que adivinar un número natural $x > 0$ mediante preguntas del tipo "¿es $x < y$?", y lo que tenemos que optimizar es el número de preguntas. Como este juego es demasiado fácil para el algoritmo óptimo si le permitimos que conozca x, diremos que la ventaja que tiene es que le permitimos hacer una primera pregunta para que conozca *la cantidad de bits* ℓ que requiere representar x. Conociendo ℓ, el óptimo sabe que $x \in [2^{\ell-1}..2^\ell - 1]$ y entonces puede hacer búsqueda binaria en el rango, a costo $\ell - 1$. En total, el óptimo realiza entonces ℓ preguntas.

El algoritmo en línea, que no puede preguntar por ℓ, puede buscarlo "secuencialmente", realizando preguntas sucesivas de la forma: ¿es $x < 2^k$? para $k = 1, 2, \ldots$ Una vez determinado ℓ con ℓ preguntas, completa con búsqueda binaria en $[2^{\ell-1}..2^\ell - 1]$, la cual requiere otras $\ell - 1$ preguntas. El costo total, $2\ell - 1$, es entonces una 2-aproximación.

Esta 2-aproximación se corresponde con un método de representación de enteros llamado *codificación* γ, que es útil para cuando debemos representar un número cuya magnitud desconocemos, y queremos usar un espacio cercano al necesario. La codificación γ contiene

básicamente las respuestas a las preguntas que realizamos: sí es 0 y no es 1, y la primera pregunta del método óptimo la representamos con un 1. Por ejemplo, si el método óptimo busca $x = 5 = 101_2$, primero determina que usa 3 bits (y ponemos un 1), es decir que está en $[4..7]$. La primera pregunta que hará será ¿es $x < 6$? La respuesta es sí $= 0$. Luego preguntará ¿es $x < 5$? La respuesta es no $= 1$, y con ello determina $x = 5$. La secuencia de respuestas es 101, la representación binaria de 5.

Ahora bien, el algoritmo en línea debe comenzar preguntando ¿es $x < 2^1$? no $= 1$, ¿es $x < 2^2$? no $= 1$, ¿es $x < 2^3$ sí $= 0$. Ahora sabe que $x \in [4..7]$ y realiza las mismas dos preguntas de la búsqueda binaria. La codificación es entonces $\gamma(5) = 110\,01$.

Resumiendo, como $\ell = 1 + \lfloor \log_2 x \rfloor$, la codificación $\gamma(x)$ usa $2\lfloor \log_2 x \rfloor + 1$ bits. Primero escribe $\lfloor \log_2 x \rfloor$ 1s seguidos de un 0, y luego los $\lfloor \log_2 x \rfloor$ bits más bajos de la representación de x (es decir, omite el bit más alto, ya que siempre es un 1).

Búsqueda exponencial. Supongamos que queremos buscar un número x en un arreglo ordenado $A[1..n]$, pero sospechamos que x está cerca del comienzo. En vez de una búsqueda binaria de costo $O(\log n)$, quisiéramos encontrar x a costo $O(\log m)$, donde $A[m] = x$. Claro que no conocemos m de antemano.

Lo que hacemos entonces es comparar x con $A[2^k]$ para $k = 0, 1, 2, \ldots$ hasta encontrar el primer k tal que $A[2^k] \geq x$. Luego de estas k preguntas, sabemos que $2^{k-1} < m \leq 2^k$, con lo cual completamos la búsqueda binaria con otras $k - 1$ comparaciones. El costo total de la búsqueda es entonces $2k - 1 = O(\log m)$, como deseábamos.

Suponga que tenemos que buscar r valores $x_1 < x_2 < \ldots < x_r$ en A. Si buscáramos cada uno con búsqueda binaria, el costo total sería $O(r \log n)$. En cambio, si usamos búsqueda exponencial a partir del punto donde encontramos el elemento anterior, el costo total será $O(\sum \log(m_i - m_{i-1} + 1))$, donde m_i es el lugar donde se encuentra x_i, y $m_0 = 0$. Se puede probar que esta suma se maximiza cuando los m_i están equiespaciados, en cuyo caso la suma es $O(r \log \frac{n}{r})$. El costo por elemento va tendiendo a constante cuando se buscan muchos elementos.

Por ejemplo, para intersectar dos listas crecientes $x_1, \ldots, x_r$ e $y_1, \ldots, y_n$, suponiendo que $r \leq n$, tenemos dos opciones: o un merge que recorre secuencialmente las dos listas, a costo $O(r + n) = O(n)$, o buscar cada elemento x_i en la lista de las y, a costo $O(r \log n)$. Con búsqueda exponencial, podemos aplicar siempre el segundo método, de costo $O(r \log \frac{n}{r})$, obteniendo siempre la mejor de las dos complejidades (y frecuentemente mejor que ambas).

Codificación δ. Cuando queremos representar números x algo mayores, buscar ℓ en forma secuencial como en los códigos γ puede ser demasiado lento. Podríamos en vez usar búsqueda exponencial para ℓ. En vez de preguntar ¿es $x < 2^k$? preguntaremos ¿es $\ell < 2^k$?, o lo que es lo mismo, ¿es $x < 2^{2^k - 1}$?. La búsqueda exponencial de ℓ requerirá $2\lfloor \log_2 \ell \rfloor + 1$ preguntas, y luego haremos las $\ell - 1$ preguntas finales para determinar x. Por lo tanto, en vez de necesitar el doble de preguntas que el óptimo necesitamos un factor extra de $O(1 + \frac{\log \ell}{\ell}) = O(1 + \frac{\log \log x}{\log x})$, que va mejorando a medida que representamos números mayores.

Esta estrategia se corresponde con otra codificación llamada δ. El código $\delta(x)$ está formado por el código $\gamma(\lfloor \log_2 x \rfloor + 1)$ seguido de los últimos $\lfloor \log_2 x \rfloor$ bits de x. Por lo tanto su largo es de $\lfloor \log_2 x \rfloor + 2\lfloor \log_2(\lfloor \log_2 x \rfloor + 1) \rfloor + 1 = \lfloor \log_2 x \rfloor + O(\log \log x)$ bits.

Si bien no se usa en la práctica, se podría iterar con esta técnica para diseñar códigos que usen $\lfloor \log_2 x \rfloor + \lfloor \log_2(\lfloor \log_2 x \rfloor + 1) \rfloor + O(\log \log \log x)$ bits, etc.

5.1.2. Búsqueda en la línea

Supongamos que un robot debe encontrar un objeto a lo largo de una línea, pero no sabe en cuál de las dos direcciones está. Si el objeto está a distancia d, entonces el algoritmo óptimo, que conoce la dirección, lo encuentra en tiempo d. El algoritmo en línea, en cambio, debe usar una estrategia de ir exporando zonas cada vez mayores en una dirección y luego en otra, hasta dar con el objeto.

Nuevamente, consideraremos una estrategia exponencial, en que el robot camina en una dirección c^0, en la otra c^1, luego otra vez en la primera c^2, etc. Supongamos primero que el objeto está en la dirección en que se movió por primera vez. Las posiciones a las que va llegando el robot luego de los pasos pares son $c^0, c^0 - c^1 + c^2, \ldots$, lo que en el paso $2k$ es

$$P(2k) \quad = \quad \sum_{i=0}^{k} c^{2i} - \sum_{i=0}^{k-1} c^{2i+1} \quad = \quad \frac{c^{2k+1}+1}{c+1}.$$

El costo total de los pasos 0 a k es $C(k)$, con

$$C(k) \quad = \quad \sum_{i=0}^{k} c^{i} \quad = \quad \frac{c^{k+1}-1}{c-1}.$$

Ahora bien, lo peor que puede pasar es que el objeto esté en una posición de la forma $P(2k)+1$, pues en ese caso el robot realizará hasta el paso $2k$, luego realizará el paso $2k+1$ hacia atrás, de largo c^{2k+1}, luego volverá esa distancia de c^{2k+1}, y recién en la siguiente celda encontrará el objeto. El costo total será entonces

$$C(2k+1) + c^{2k+1} + 1 \quad = \quad \frac{(2c-1)c^{2k+1} + c - 2}{c-1},$$

mientras que el costo del algoritmo óptimo será $P(2k)+1$. El cociente, convirtiendo pesimistamente el costo del óptimo a $\frac{c^{2k+1}}{c+1}$ y descartando $\frac{c-2}{c-1}$ del costo del algoritmo en línea (pues está acotado por la constante $b = 1$), se reduce a

$$\frac{(2c-1)(c+1)}{c-1},$$

el cual se minimiza para $c = 2$ (es decir, la mejor estrategia es buscar en potencias de 2) y nos arroja una competitividad de 9. Nuevamente, es claro que sólo la estrategia exponencial

permite una competitividad constante. El costo de no saber en qué dirección buscar es un factor de 9, sorprendentemente alto. Si el objeto se encuentra hacia el otro lado, tendremos

$$P(2k+1) \quad = \quad -\frac{c^{2k+2}-1}{c+1}$$

y el objeto estará en el peor caso en una posición de la forma $P(2k+1)-1$, la que encontraremos a costo

$$C(2k+2)+c^{2k+2}+1 \quad = \quad \frac{(2c-1)c^{2k+2}+c-2}{c-1},$$

mientras que el costo del óptimo será $-P(2k+1)+1$. Simplificando, llegamos al mismo resultado que antes.

Existen versiones más complejas de este problema en dos dimensiones, con aplicaciones más reales a robótica.

5.2. Paginamiento

Un caso emblemático de problema en línea ocurre cuando los sistemas operativos implementan una memoria virtual mayor a la física. La memoria virtual se almacena en disco y sólo algunas de sus páginas residen en la memoria física, pues ésta es menor y está usualmente llena. Cuando un proceso pide acceder a una página que no está en la memoria física (esto se llama *fallo de página*), se debe elegir una página *víctima* de la memoria física para devolverla al disco y así hacer lugar para traer la página deseada a la memoria física.

Un algoritmo que conociera los pedidos futuros de páginas elegiría como víctima aquella página que falte más tiempo para que vuelva a ser solicitada. En la realidad, debemos usar un algoritmo en línea y deseamos establecer su competitividad.

Analizaremos un algoritmo llamado *Least Recently Used (LRU)*. Este algoritmo elige como víctima la página que hace más tiempo que no se accede. Para analizar el LRU, consideraremos una memoria física de k páginas.

Mostraremos que LRU es k-competitivo. Para ello, consideremos una secuencia de n pedidos de páginas, y cortémosla en *segmentos* de k fallos de página del algoritmo LRU. Vamos a mostrar que el algoritmo óptimo debe fallar al menos una vez en cada segmento, con lo cual la k-competitividad quedará establecida.

Todo segmento termina en un fallo de página. Sea A la última página pedida en el segmento anterior, la cual ha producido un fallo de página (el k-ésimo de ese segmento). Existen tres posibilidades para el segmento actual:

En algún momento vuelve a fallar en A. Como A era la página más reciente cuando empezó el segmento, si dentro del segmento se vuelve a fallar en A es porque se la había elegido como víctima antes en el segmento. En ese momento, entonces, A era la

página usada menos recientemente de las k que había en memoria física. Por lo tanto, se debieron mencionar otras k páginas, distintas a A y distintas entre sí, para que al mencionar la k-ésima A hubiera sido la más antigua y se la eligiera como víctima. Al fallar ahora en A, tenemos que el segmento menciona $k+1$ páginas distintas. En este caso, el algoritmo óptimo está obligado a fallar al menos una vez.

Se falla 2 veces en una página $B \neq A$. En el período que transcurre entre los dos fallos de B se puede razonar exactamente como con A en el punto anterior: se deben mencionar $k+1$ páginas distintas en ese período.

Ninguna de las anteriores. Como se falla k veces sin repetir ninguna B ni fallar en A, deben mencionarse k páginas distintas que no son A. Como A estaba en la memoria física al comenzar, el óptimo también debe fallar al menos una vez.

Optimalidad. Ningún algoritmo en línea puede ser mejor que k-competitivo. Dado cualquier algoritmo en línea, un adversario puede siempre pedir a continuación la página que el algoritmo acaba de sacar de la memoria física, haciéndolo así fallar n veces. El algoritmo óptimo, en cambio, decide qué hacer una vez que ha visto la secuencia de pedidos (por lo tanto no puede haber un adversario que lo obligue a fallar siempre). Como el óptimo elige la página que falta más tiempo para que vuelva va pedirse, cada vez que falla tiene garantizado que no volverá a fallar en los siguientes $k-1$ pedidos. Por lo tanto el óptimo nunca falla más de $\frac{n}{k}$ veces, incluso en la secuencia que hace fallar n veces al algoritmo en línea. En ese sentido, LRU es lo mejor que puede esperarse en términos de peor caso.

Otro esquema k-competitivo. El esquema *First In, First Out (FIFO)* selecciona como víctima la página que hace más tiempo que entró a la memoria física (sin considerar cuándo fue accedida después de entrar). Un análisis casi idéntico (y algo más simple) del que hicimos para LRU muestra que FIFO también es k-competitivo.

Esquemas no competitivos. Los esquemas *Most Recently Used (MRU)* y *Last In, First Out (LIFO)* no son competitivos. Considere que tenemos en la memoria $k-2$ páginas que no mencionaremos, y las páginas A y B. Si el proceso pide A y luego B, entonces este esquema sacará A de la memoria, por ser la más reciente. Luego el proceso pide A y este esquema saca B de la memoria por ser el más reciente. El proceso entonces pide B, y así. En n pedidos podemos tener n fallos, cuando el óptimo habría mantenido A y B en la memoria, fallando $O(1)$ veces.

5.3. Move to Front

En el capítulo de amortización, mostramos que los splay trees obtienen un costo amortizado de $3H+1$ por operación, donde H era la entropía de las frecuencias de búsqueda. Como

un árbol construido por un algoritmo que conozca esas frecuencias requiere $H + 2$ operaciones, los splay trees son 3-competitivos. Estudiaremos ahora una versión análoga que puede utilizarse cuando no existe un orden total entre los elementos, pues sólo requiere compararlos por igualdad.

Supongamos que tenemos un arreglo $A[1..n]$ de elementos (sin un orden) en el que buscamos repetidamente mediante acceder $A[1]$, $A[2]$, ... hasta dar con el $A[k] = x$ buscado. Como podemos poner los elementos en cualquier orden y el costo de acceder a $x = A[k]$ es k, queremos ordenar el arreglo en orden decreciente de frecuencia de acceso, para minimizar el costo de todas las búsquedas. El problema es que los accesos aún no han ocurrido, por lo cual tenemos un problema en línea.

Lo que haremos es partir con un orden arbitrario y usar la estrategia *move to front (MTF)*, es decir, el elemento que se accede se reubica en el comienzo de A. Más precisamente, si $A[1..n] = x_1, x_2, \ldots, x_n$ antes de buscar $x = x_k$, entonces luego de encontrarlo reescribiremos $A[1..n] = x_k, x_1, x_2, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n$. Si medimos el costo en términos de lecturas en A, entonces esta reubicación cuesta $k - 1$ lecturas adicionales, para mover $A[1..k-1]$ a $A[2..k]$. Ignoraremos, sin embargo, este costo, pues puede evitarse implementando A como una lista enlazada o moviendo los elementos a medida que buscamos x (note que, en todo caso, el costo sólo se duplicaría, de modo que obtendríamos competitividad con el doble de la constante).

Si bien suena razonable como estrategia (análoga a los splay trees, que envían a la raíz al nodo accedido), nos preguntamos si MTF es competitivo contra un algoritmo que conociera de entrada las frecuencias y dejara el arreglo ordenado en forma óptima, es decir, por frecuencias decrecientes, en un arreglo fijo $A_{OPT}[1..n]$.

Para analizar el MTF, usaremos análisis amortizado con función potencial. En cualquier momento de la ejecución de las búsquedas, llamaremos *inversión* a cada par de elementos de A que no está en el mismo orden en A_{OPT}. Es decir, cada par $A[i] = x$ y $A[j] = y$, con $i < j$, que están en $A_{OPT}[i'] = y$ y $A_{OPT}[j'] = x$ con $i' < j'$, es una inversión (supondremos que todos los elementos son distintos). Note que las inversiones cuentan *todos* los pares desordenados, no sólo los consecutivos. Definiremos entonces la función potencial

$$\Phi \quad = \quad \text{número total de inversiones.}$$

Si el elemento buscado x está en $A_{OPT}[i]$ y en $A[k]$, entonces el algoritmo óptimo tiene costo i y el MTF tiene costo k. A esto debemos sumar el cambio del potencial Φ cuando MTF mueve x al comienzo de A. Lo primero a notar es que Φ sólo cambia con respecto a los pares que involucran a x y a algún elemento y en $A[1..k-1]$. Luego de mover x a $A[1]$, éste aparecerá antes de y. ¿Esto aumenta o disminuye Φ? Depende:

- Si y está antes que x en A_{OPT}, entonces hemos creado una inversión que no existía antes, lo cual incrementa Φ en 1.
- Si y está después de x en A_{OPT}, entonces hemos destruido una inversión que existía antes, lo cual decrementa Φ en 1.

Esto significa que al mover x podemos crear a lo sumo $i-1$ inversiones nuevas, para aquellos y en $A_{OPT}[1..i-1]$. Digamos que creamos $0 \leq r \leq i-1$ inversiones. Entonces los otros $k-1-r \geq k-i$ elementos y que cambian su relación con x destruyen inversiones. Sumando las inversiones que se crean y restando las que se destruyen, tenemos para la j-ésima operación

$$\Delta\Phi_j \;=\; r-((k-1)-r) \;=\; 2r-(k-1) \;\leq\; 2i-2-(k-1) \;=\; 2i-k-1.$$

Por otro lado, el costo real c_i de la búsqueda es, como dijimos, k, lo que nos da un costo amortizado de

$$c_j+\Delta\Phi_j \;=\; k+(2i-k-1) \;<\; 2i,$$

es decir, 2 veces el costo del óptimo.

Para concluir, hemos analizado el paso de una operación a la siguiente, pero el valor de Φ_0 puede llegar a ser $\frac{n(n-1)}{2}$. Si consideramos el largo n del arreglo como constante (y un número variable m de búsquedas), entonces podemos decir que MTF es 2-competitivo, pues en cualquier secuencia tenemos que los costos totales satisfacen

$$m\cdot c_{MTF} \;\leq\; m\cdot 2\,c_{OPT} \;+\; \frac{n(n-1)}{2},$$

es decir,

$$c_{MTF} \;\leq\; 2\,c_{OPT} \;+\; \frac{n(n-1)}{2m}.$$

Si no consideramos n constante, entonces debemos establecer que deben realizarse $m=\Omega(n^2)$ operaciones para que MTF sea 2-competitivo. Es decir, las suficientes para que se extinga el efecto de un orden inicial poco conveniente en A.

Óptimo dinámico. ¿Qué ocurre si le permitimos al óptimo que también mueva elementos en el arreglo, dado que puede ser que el orden óptimo en un momento del tiempo sea muy distinto del de otro momento? No podemos permitirle que mueva cualquier elemento a cualquier posición a costo $O(1)$, pues entonces en cada paso pondría en $A_{OPT}[1]$ el elemento que se buscará a continuación. Es más interesante que le permitamos intercambiar elementos adjacentes, de modo que mover un elemento de la posición i a la j le cueste $|i-j|$.

En este modelo, el movimiento que hace MTF de $A[k]$ a $A[1]$ también puede ejecutarse con este tipo de intercambios, a costo $k-1$. Si rehacemos el análisis de MTF considerando que su costo para buscar $x=A[k]$ no es k sino $2k-1$, y usando como función potencial

$$\Phi \;=\; 2\cdot(\text{número de inversiones}),$$

obtendremos $c_j+\Delta\Phi_j \leq 4i$, es decir, una 4-aproximación.

Supongamos entonces que el óptimo, luego de encontrar x en $A_{OPT}[i]$, decide intercambiar w pares consecutivos de A_{OPT} en forma arbitraria. El costo pagado por el algoritmo óptimo

será entonces $i+w$. El efecto en nuestro análisis es que esos intercambios pueden hacer crecer Φ. En particular, cada intercambio en A_{OPT} puede crear una nueva inversión, de modo que los w intercambios pueden hacer crecer Φ en $2w$. En total, nuestro costo amortizado de $4i$ puede crecer entonces a $4i+2w$. Nuestro algoritmo sigue siendo entonces 4-competitivo, pues

$$\frac{4i+2w}{i+w} \quad \leq \quad 4.$$

Menos que MTF. ¿Es necesario trabajar tanto como lo hace MTF luego de cada búsqueda para lograr competitividad? Consideremos una versión que, al encontrar un elemento $x = A[k]$, solamente lo intercambia con $A[k-1]$ (si $k > 1$), moviéndolo así de a una posición hacia el comienzo del arreglo. Es fácil ver que esta versión no es competitiva, pues si $A[n] = x$ y $A[n-1] = y$ son el último y el penúltimo elementos del arreglo al comenzar, entonces haremos que el algoritmo pague n por cada operación mediante buscar x, y, x, y, ... El óptimo, en cambio, pagará 1 ó 2 mediante poner a x y a y al comienzo de A_{OPT}.

Note que, en el caso de MTF, también podemos diseñar una secuencia de búsquedas donde MTF pague siempre n, haciéndolo que busque el elemento que en cada momento está al final del arreglo. La diferencia es que, por lo que hemos demostrado, esa secuencia tampoco es buena para el algoritmo óptimo: lo obliga a pagar al menos $\frac{n}{2}$ lecturas en cada búsqueda. La esencia de demostrar competitividad no es mostrar que no hay secuencias malas para el algoritmo en línea, sino mostrar que esas secuencias también son malas para un algoritmo óptimo que conociera el futuro.

Compresión usando MTF. El MTF se usa también como método de compresión. La idea es que se parte con los n símbolos distintos en un orden arbitrario pero conocido en $A[1..n]$. Luego, para codificar el símbolo x, se lo busca en A, y si se lo encuentra en $A[k] = x$, se emite el número k, moviendo luego x al frente del arreglo. El descompresor, al recibir k, decodifica $x = A[k]$ y también lo mueve al frente del arreglo.

Tal como el costo k de las búsquedas se reduce al mover los elementos al frente de A, los números k emitidos por este compresor tienden a ser pequeños. Por ejemplo, si los codificamos con los códigos-δ que vimos antes en este capítulo, requeriremos $\log_2 k + O(\log\log k)$ bits. ¿Esta compresión basada en MTF nos ofrece alguna garantía de optimalidad?

Consideremos primero un compresor estático, es decir, que le asigna el mismo código a x cada vez que lo emite. En el capítulo de cotas inferiores vimos que el largo total en bits de los códigos con un codificador óptimo que conozca la frecuencia $f(x)$ con que se emite el elemento x no puede ser inferior a la entropía:

$$m\,H \;=\; \sum_x f(x)\log_2\frac{m}{f(x)}.$$

Consideremos lo que ocurre con MTF. El elemento x se emite $f(x)$ veces, digamos que en los instantes $1 \leq t_1 < t_2 < \ldots < t_{f(x)} \leq m$. Note que, entre el momento t_{i-1} y t_i, se emitieron

precisamente $t_i - t_{i-1} - 1$ símbolos distintos de x. Si todos ellos fueran distintos entre sí, como el MTF los envió al comienzo de A, el elemento x estaría en la posición $t_i - t_{i-1}$ de A, y si no, estaría antes. Esto significa que codificar x en el instante t_i nos cuesta a lo sumo $\log_2(t_i - t_{i-1}) + O(\log\log(t_i - t_{i-1}))$ bits. El costo de codificar todas las ocurrencias de x es entonces a lo sumo

$$\log_2 n + O(\log\log n) + \sum_{i=2}^{f(x)} \log_2(t_i - t_{i-1}) + O(\log\log(t_i - t_{i-1}))$$

bits, donde estamos suponiendo lo peor al codificar t_1 porque no sabemos dónde estaba x en A. La suma se maximiza cuando todos los t_i están equiespaciados, $t_i = i\,\frac{m}{f(x)}$, llegando a

$$\log_2 n + O(\log\log n) + (f(x) - 1)\left(\log_2 \frac{m}{f(x)} + O\left(\log\log \frac{m}{f(x)}\right)\right)$$

bits. Sumando sobre todos los n elementos x obtenemos

$$n\log_2 n + O(n\log\log n) + \sum_x f(x)\log_2 \frac{m}{f(x)} + O\left(f(x)\log\log \frac{m}{f(x)}\right),$$

lo cual se puede mostrar que es a lo más

$$m\,H + m\,O(\log H) + O(n\log n).$$

Es decir, el MTF se acerca a los H bits de entropía H por símbolo emitido, con un costo extra de $O(\log H)$ bits por símbolo, y un costo extra total de $O(n\log n)$ bits. Este último costo tiene que ver con el reordenamiento original de A, e indica que deben emitirse $\Omega(n\log n)$ símbolos para que este costo extra sea $O(1)$ por símbolo. En este caso, podemos decir que el compresor MTF es $(1 + O(\frac{\log H}{H}))$-competitivo contra un compresor estático óptimo.

¿Qué ocurre si el compresor óptimo puede ser dinámico, cambiando los códigos a medida que va emitiendo los símbolos? Supongamos que permitimos que, cuando lo desee, el algoritmo óptimo gaste $n\log_2 n$ bits en establecer un nuevo orden de los elementos. Esto divide el tiempo en t períodos de largos $m_1, \ldots, m_t$. Podemos aplicar el análisis de la compresión MTF dentro de cada período, obteniendo la suma de las entropías locales, $m_i\,H_i + m_i\,O(\log H_i)$, más los extras $O(tn\log n)$. El óptimo dinámico, en cambio, costará $m_i\,H_i + (t-1)n\log n$. La competitividad de MTF se mantiene igual.

5.4. Los k Servidores

Un problema en línea importante es el de tener k *servidores* que deben movilizarse para atender *pedidos* que ocurren en un determinado lugar, por ejemplo policía, ambulancias, bomberos, taxis, repartidores de pizza, etc. El objetivo es minimizar desplazamiento total

de los servidores para cubrir los pedidos que van apareciendo a lo largo del tiempo, pero no se conocen de antemano los pedidos futuros. Queremos una técnica que sea competitiva con un algoritmo óptimo que conozca todos los pedidos de antemano.

Consideremos una versión simplificada en la que los k servidores están en una línea. Un algoritmo obvio parece ser desplazar el servidor más cercano al pedido, para minimizar el recorrido total. Sin embargo, consideremos el caso de dos servidores, uno lejano y otro cercano a dos puntos x e y donde ocurren pedidos alternadamente. Con esta estrategia, elegiremos el servidor cercano para cubrir todos los pedidos, de modo que cada pedido nos costará un desplazamiento de $|x-y|$, mientras que el lejano no se usará. El algoritmo óptimo, en cambio, mueve un servidor a x y el otro a y y luego no necesita desplazarse más.

Una estrategia que demostraremos competitiva es la siguiente: sean $s_1 \leq s_2 \leq \ldots \leq s_k$ las posiciones de los servidores en la línea. Entonces

- Si el pedido ocurre en una posición $x < s_1$, desplazamos el servidor 1 a x, a costo $s_1 - x$.
- Si el pedido ocurre en una posición $x > s_k$, desplazamos el servidor k a x, a costo $x - s_k$.
- Si el pedido ocurre en una posición $s_i \leq x \leq s_{i+1}$, desplazamos *ambos* servidores, i e $i+1$, en direcciones opuestas hacia x, hasta que el más cercano lo alcance. Es decir, ambos se mueven mín$(x - s_i, s_{i+1} - x)$. Note que en el caso particular en que $x = s_i$, nadie se moverá e i atenderá el pedido sin costo.

Demostraremos la k-competitividad de este esquema. Para ello, llamaremos $o_1 \leq o_2 \leq \ldots \leq o_k$ a las posiciones de los servidores del algoritmo óptimo, y definiremos la función potencial

$$\Phi \quad = \quad \left(\sum_{1 \leq i < i' \leq k} s_{i'} - s_i \right) \; + \; k \cdot \left(\sum_{1 \leq i \leq k} |s_i - o_i| \right).$$

Observe que los servidores del algoritmo en línea siempre mantienen el orden $s_i \leq s_{i+1}$, y los del óptimo tampoco necesitan cambiar de identidad para mantener el orden $o_i \leq o_{i+1}$: una estrategia donde i se mueve a la derecha hasta rebasar a $i+1$ se puede cambiar por otra donde i alcanza a $i+1$ y luego $i+1$ continúa a la derecha haciendo lo que habría hecho i. A cambio, debemos permitir que el óptimo realice varios movimientos de servidores cuando llega un pedido x. Consideraremos primero, según dónde cae x, el costo de los movimientos del algoritmo en línea y el movimiento del óptimo que atiende el pedido. Al final consideraremos otros movimientos del algoritmo óptimo.

Caso $x < s_1$. Movemos s_1 hacia x, a costo $c_j = a = s_1 - x$. ¿En cuánto cambia la función potencial? La primera sumatoria aumenta en $(k-1) \cdot a$. La segunda depende de qué servidor del óptimo atienda el pedido.

Si el óptimo mueve su servidor 1**.** En este caso, el costo del óptimo será $b = |o_1 - x|$. La distancia entre s_1 y o_1, que originalmente era $|s_1 - o_1| \geq |a - b|$, ahora se hará 0, por lo cual Φ decrecerá al menos en $k \cdot |a - b|$. En total, usando que $|a - b| \geq a - b$, tendremos un costo amortizado de

$$c_j + \Delta\Phi_j \quad \leq \quad a + (k-1) \cdot a - k \cdot |a - b| \quad = \quad k \cdot (a - |a - b|) \quad \leq \quad k \cdot b,$$

lo cual es a lo sumo k veces el costo b del óptimo.

Si el óptimo mueve su servidor $i > 1$**.** En este caso, el costo del óptimo será $b = |o_i - x|$. La distancia entre s_1 y o_1 se reduce en a, pues debe ser $o_1 < x$ para que el servidor i pueda moverse hasta x sin cruzarse con el servidor 1. En cambio, la distancia entre los dos servidores i puede crecer hasta en b. En total, tendremos un costo amortizado de

$$c_j + \Delta\Phi_j \quad \leq \quad a + (k-1) \cdot a + k \cdot (b - a) \quad = \quad k \cdot b,$$

lo cual es a lo sumo k veces el costo b del óptimo.

Caso $s_i \leq x \leq s_{i+1}$**.** El algoritmo en línea moverá ambos servidores, i e $i+1$, hacia x, una distancia de $a = \text{mín}(x - s_i, s_{i+1} - x)$. El costo será entonces $c_j = 2 \cdot a$. No es difícil ver que, en la primera sumatoria de Φ, los efectos de mover i e $i+1$ contra otros i' se cancelan, y sólo queda la diferencia $s_{i+1} - s_i$, que decrece en $2 \cdot a$. Con respecto a la segunda sumatoria y los movimientos del óptimo, tenemos dos subcasos.

Si $o_{i+1} < x$ **o** $x < o_i$**.** En el primer caso, el acercamiento de s_{i+1} a x reduce la distancia $|s_{i+1} - o_{i+1}|$ en a, mientras que en el segundo caso, el acercamiento de s_i a x reduce la distancia $|s_i - o_i|$ en a. En ambos casos, el movimiento del otro servidor del algoritmo en línea puede incrementar la distancia con el correspondiente servidor del óptimo en a lo sumo a. Estos efectos entonces se cancelan en la segunda sumatoria de Φ, dándonos un costo amortizado de

$$c_j + \Delta\Phi_j \quad \leq \quad 2 \cdot a + (-2 \cdot a) + k \cdot (a - a) \quad = \quad 0.$$

Por ello, el movimiento que realice el óptimo para cubrir el pedido (que aún no habíamos incluído) lo podemos considerar en forma independiente, de acuerdo al último caso considerado en esta lista.

Si $o_i \leq x \leq o_{i+1}$**.** Para que no se crucen los servidores del óptimo, éste debe usar el servidor i o el $i+1$ para mover hacia x. Supongamos que mueve el i a costo $b = x - o_i$; el otro caso es simétrico. Independientemente de qué servidor del algoritmo llegue a x, ambos servidores i se mueven en la misma dirección, el óptimo en b unidades y el algoritmo en a. Por ello, la distancia $|s_i - o_i|$ decrece en $|a - b|$. Por otro lado, el servidor $i+1$ del algoritmo en línea se mueve a unidades

en la otra dirección, posiblemente alejándose del servidor $i+1$ del óptimo. En total, usando que $|a-b| \geq a-b$, tenemos

$$c_j + \Delta\Phi_j \quad \leq \quad 2 \cdot a + (-2 \cdot a) + k \cdot (a - |a-b|) \quad \leq \quad k \cdot b,$$

que es k-competitivo contra el costo b del óptimo.

Caso $x > s_k$. Análogo al caso $x < s_1$.

Otros movimientos del óptimo. Además del último movimiento que consideramos, que finalmente cubre el pedido, el óptimo puede mover todos los servidores que desee, como explicamos. Cualquier movimiento de un servidor óptimo i a una distancia b puede incrementar Φ en $k \cdot b$ debido a la segunda sumatoria, pero eso aún es k-competitivo con el costo b que pagó el algoritmo óptimo para mover el servidor.

Existen algoritmos k-competitivos para otras variantes más generales del problema, y se cree que esto es posible para k servidores en un espacio métrico cualquiera, pero hasta ahora sólo se conocen algoritmos $(2k-1)$-competitivos para este caso más general.

5.5. Ficha Resumen

- Códigos γ y δ, búsqueda exponencial e intersección de listas.
- Búsqueda de un robot en la línea: 9-competitivo.
- Paginamiento: k-competitivo.
- Move-to-front: 2-competitivo contra un óptimo estático y 4-competitivo contra uno dinámico.
- Compresión usando move-to-front: $(1 + O(\frac{\log H}{H}))$-competitivo, donde H es la entropía.
- Servidores en la línea: k-competitivo con k servidores.

5.6. Material Suplementario

El tema de algoritmos en línea no se trata en muchos libros de texto, pero el de Borodin y El-Yaniv [BEY98] está dedicado enteramente al tema. Considera competitividad de peor caso pero también aleatorizada. Entre muchos temas que cubre, incluye algunos vistos en este capítulo: paginamiento (cap. 3), move-to-front (cap. 1), y k servidores en la línea y otros modelos más complejos (cap. 10).

Lee et al. [LTCT05, cap. 10] también dedican un capítulo a tratar algoritmos en línea en profundidad. Además de las definiciones, dedican la sección 12.2 al problema de los k

servidores, pero extendido a moverse a lo largo de las aristas de un árbol dibujado en el plano. El resto del capítulo se dedica a varios otros problemas que no describimos. En una sección anterior [LTCT05, sec. 10.3] describen brevemente el análisis amortizado de MTF contra un óptimo estático.

Otras fuentes online de interés:

- `www.cs.huji.ac.il/course/2005/algo2/on-line/on-line-course.html`
- `www.cs.cmu.edu/ avrim/451f13/lectures/lect1107.pdf`
- `web.stanford.edu/class/cs369/files/cs369-notes`
- `www14.in.tum.de/personen/albers/papers/inter.pdf`
- `www-math.mit.edu/~goemans/notes-online.ps`
- `www.youtube.com/watch?v=2RxCCEHlEys`

Capítulo 6

Algoritmos Probabilísticos y Aleatorizados

En este capítulo consideraremos algoritmos que rompen al menos una de las dos siguientes suposiciones básicas de los algoritmos clásicos:

- El algoritmo nunca se equivoca.
- El algoritmo siempre hace lo mismo frente a la misma entrada.

No cumplir con la primera suposición suena extraño a primera vista. En el mundo clásico, un algoritmo que ordena pero que a veces se equivoca, ¡simplemente no es un algoritmo que ordena! Un algoritmo *probabilístico*, en cambio, sí tiene permitido equivocarse. Generalmente, el algoritmo se equivoca con una cierta *probabilidad de error*, la que puede hacerse arbitrariamente pequeña con un incremento moderado en el costo del algoritmo. En muchos casos, es muy barato hacer que la probabilidad sea más pequeña que la de un fallo del hardware, por ejemplo, de modo que en términos prácticos un buen algoritmo probabilístico puede ser completamente satisfactorio. Por otro lado, muchos problemas que son sumamente costosos de resolver sin error pueden ser resueltos muy eficientemente si se permite una pequeñísima probabilidad de equivocarse.

Los algoritmos que hacen siempre lo mismo se llaman *determinísticos*, y los que no, se llaman *aleatorizados*. Estos últimos incorporan una componente de azar en su ejecución, de modo que pueden no hacer lo mismo frente a la misma entrada. Note que estos algoritmos pueden tener un costo distinto cada vez que ejecutan frente al mismo input, por lo cual más que un costo fijo para cada input, su costo es una distribución de probabilidad. Esta distribución puede no tener que ver con la distribución de la entrada, sino con la de las decisiones aleatorias que toma internamente el algoritmo. Aparece entonces la noción de *costo esperado* del algoritmo, que no es igual al costo promedio. El costo esperado promedia sobre las distintas ejecuciones posibles del algoritmo frente a un input, y considera el peor input posible (aunque el promedio suele ser independiente del input). El costo promedio, en

cambio, se refiere a los distintos inputs posibles, para cada uno de los cuales un algoritmo determinístico tiene un costo fijo.

El interés de los algoritmos aleatorizados reside en que pueden ofrecer un costo esperado independiente de cualquier suposición sobre cómo se distribuyen los inputs, lo que constituye una medida mucho más robusta que el costo promedio, que siempre depende de la distribución del input. En particular, a un algoritmo determinístico de buen costo promedio se lo puede hacer comportar sistemáticamente mal dándole "malos inputs", mientras que para un algoritmo aleatorizado de buen costo esperado no existen "malos inputs", sino "malas ejecuciones": si se produce una ejecución muy costosa, lo más probable es que volviendo a correr incluso sobre el mismo input no vuelva a ocurrir lo mismo. Esto además hace a la aleatorización una buena herramienta contra adversarios maliciosos, que eligen el input que haga fallar un algoritmo (por ejemplo, en criptografía o en ataques tipo denial-of-service).

Es común que los algoritmos probabilísticos sean también aleatorizados, en cuyo caso la probabilidad de error no depende de la distribución de la entrada, sino que vale para cualquier input. De hecho, en la literatura se suele englobar a ambos, generalmente bajo el término de *randomized algorithms*.

Los computadores no producen números aleatorios, sino secuencias pseudoaleatorias. Estas secuencias son realmente determinísticas, pero están diseñadas para superar varios tests estadísticos de aleatoriedad. Es conveniente además partir de un punto de esta secuencia (llamado la semilla) determinado por algo impredecible, como los dígitos más bajos del reloj del computador, para evitar que el programa reciba siempre la misma secuencia. Todos los lenguajes ofrecen acceso a estos generadores aleatorios, y también se pueden programar con bastante facilidad.

Comenzaremos con un conjunto de definiciones relacionadas con los algoritmos probabilísticos y aleatorizados, y varios ejemplos muy simples. Luego veremos ejemplos más importantes de algoritmos y estructuras de datos que usan la aleatoriedad y permiten el error.

6.1. Definiciones y Ejemplos Simples

6.1.1. Algoritmos tipo Monte Carlo y Las Vegas

Un algoritmo probabilístico es de tipo *Monte Carlo* si puede entregar una respuesta equivocada. Algunos algoritmos Monte Carlo pueden verificar si su respuesta es correcta o no a un costo razonable. Cuando pueden hacerlo y son algoritmos aleatorizados, es posible reejecutarlos varias veces, para reducir la probabilidad de que entreguen una respuesta incorrecta. Incluso es posible reejecutarlos indefinidamente, hasta que entreguen una respuesta correcta. Este último tipo de algoritmo se llama *Las Vegas*, el cual nunca se equivoca pero no tiene un tiempo de peor caso garantizado, sólo un tiempo esperado acotado. Sin embargo, incluso con un algoritmo de tipo Monte Carlo que no pueda verificar su respuesta, es posible reducir su probabilidad de error arbitrariamente mediante repetirlo una cantidad fija de veces y

devolver la "mejor" respuesta de las obtenidas. Veamos algunos ejemplos simples.

Un pez grande. Considere el problema de pescar un pez grande en el océano, donde definimos "grande" como "mayor o igual que la mediana". Un algoritmo determinístico debe pescar al menos $\lceil \frac{n+1}{2} \rceil$ peces para garantizar que el máximo de ellos es grande, donde n es el número de peces en el océano. Consideremos ahora el expediente simple de sacar un pez y declararlo grande. Este es un algoritmo de Monte Carlo que se equivoca con probabilidad $\frac{1}{2}$, y ni siquiera podemos saber (a un costo razonable) si nos equivocamos o no. Si bien no parece un gran logro, considere iterar este algoritmo: pescamos k peces y nos vamos quedando con el mayor, que finalmente declaramos grande. Este es un algoritmo de costo $O(k)$, y la probabilidad de equivocarnos es la de que las k veces hayamos sacado un pescado pequeño, es decir $\frac{1}{2^k}$. Por ejemplo, sacando $k = 22$ pescados, la probabilidad de equivocarse es cercana a la de ganarse el Loto. Compare esto con el costo $\Theta(n)$ de cualquier algoritmo clásico.

Suponga, en cambio, que sabemos que la mediana de los pesos de los peces del océano son 20 kilogramos. Entonces nuestro algoritmo puede verificar si se equivoca o no, y podemos repetirlo hasta que no se equivoque. El número esperado de veces que debemos pescar hasta sacar un pez grande (y sin error) es 2. Sin embargo, en el peor caso, no tenemos garantía de terminar nunca (especialmente si devolvemos los peces pequeños al océano luego de pescarlos). Este es un algoritmo tipo Las Vegas.

No es difícil imaginar problemas de corte más computacional que tienen esta estructura, por ejemplo elegir un buen alumno de una lista sin tener que recorrer media lista.

Acceso a Ethernet. El protocolo de acceso a la red Ethernet funciona de esta forma. Físicamente, todos los computadores conectados pueden leer lo que todos escriben, de modo que para enviar un mensaje a otro computador se debe indicar el destinatario en el encabezamiento del mensaje. Todos leerán el encabezado y el aludido leerá el resto del mensaje. Si dos computadores deciden escribir al mismo tiempo, sin embargo, la señal se corromperá y todos lo notarán. El protocolo de escritura es, entonces, como sigue: se espera a que no haya un mensaje escrito en la red y entonces se escribe el mensaje que se desea. Luego se lee la red. Si se puede leer lo que se escribió, entonces terminamos. Si en cambio se lee un mensaje corrupto, es porque más de un computador decidió escribir al mismo tiempo. Se espera entonces que el mensaje corrupto desaparezca más un intervalo aleatorio de tiempo, y se reintenta. Este es un algoritmo probabilístico y aleatorizado, tipo Las Vegas, más eficaz y sencillo que cualquier protocolo que intentara resolver el problema determinísticamente.

Consistencia de bases de datos. Considere verificar la consistencia entre dos copias de una gran base de datos, conectadas por una red proporcionalmente lenta. No hay algoritmo que pueda garantizar que las dos copias son iguales sin esencialmente transmitir una de ellas al lugar de la otra. En la práctica, se usa un esquema de firma digital: se calcula un hash de una de ellas y se transmite a la otra, que también calcula su hash y los compara. Las

funciones de hash son determinísticas, pero si están bien diseñadas, la probabilidad de que dos bases de datos elegidas al azar tengan la misma firma de k bits es de $\frac{1}{2^k}$.

Para hacerse una idea de lo que significa esta probabilidad de error, usemos una estimación reciente de que se producen unos 75 fallos en chips de memoria por millón de horas de funcionamiento por megabit. Por lo tanto, una firma de 70 bits entrega menor probabilidad de error que la de que ocurra un fallo de hardware que afecte uno de esos 70 bits justo en el nanosegundo en que se examina.

Existen mecanismos de firmas que además no se equivocan frente a ciertos tipos de errores muy comunes, como un número limitado de inversiones de bits, o errores en ráfaga, etc. Este es un esquema determinístico (no aleatorizado) de error tipo Monte Carlo.

En el último ejemplo, si las firmas son distintas, entonces con seguridad las dos copias son inconsistentes, pero si las firmas son iguales, aún podría ser que las copias fueran distintas (con una probabilidad muy baja). Este tipo de algoritmos Monte Carlo que responden "sí" o "no" se clasifican en *one-sided error* (en que sólo se equivocan en una dirección, como en este caso) y *two-sided error* (en que ambas respuestas pueden ser erróneas).

6.1.2. Aleatorización para independizarse del input

Veamos dos ejemplos simples de aleatorizar para independizarse de las suposiciones sobre el input, o incluso defenderse de inputs maliciosos.

Búsqueda secuencial. Supongamos que tenemos una lista de n elementos que no pueden ordenarse. Buscamos un elemento x secuencialmente en la lista, y si lo hallamos en la posición i, nuestro costo fue i. Si en nuestra aplicación los elementos de la lista son las palabras distintas que vemos en un texto y vamos insertando las nuevas al final, entonces las más comunes estarán al comienzo, pues tienden a aparecer antes. Por ello, sería bueno recorrer la lista de la posición 1 a la n. Pero si en nuestra aplicación los elementos son nombres propios que aparecen en noticias, entonces es probable que volvamos a ver los más recientes, por lo que sería mejor buscar la lista desde el final. Nos conformaríamos con pagar $\frac{n+1}{2}$ en promedio, pero cualquier estrategia determinística que elijamos puede funcionar pésimamente en un determinado entorno, el cual no podemos predecir.

Una forma segura de independizarnos de cualquier suposición externa es aleatorizar: tiramos una moneda, y si sale cara, buscamos de 1 a n, si no, buscamos de n a 1. ¿Cuál es el costo esperado de encontrar el elemento x, que está en la posición i? Es i si buscamos de 1 a n (lo que hacemos con probabilidad $\frac{1}{2}$), y es $n - i + 1$ si buscamos de n a 1 (lo que hacemos con probabilidad $\frac{1}{2}$). El costo esperado es entonces

$$\frac{1}{2} \cdot i \; + \; \frac{1}{2} \cdot (n - i + 1) \quad = \quad \frac{n+1}{2},$$

¡independiente de i! Es decir, independiente de la distribución de las búsquedas.

QuickSort aleatorizado. Otro ejemplo bien conocido es el QuickSort, que tiene costo promedio $O(n \log n)$ *suponiendo* que la distribución de las posibles permutaciones de entrada es uniforme. Pero aunque elija el pivote al medio, al principio, al final, etc., hay inputs que lo hacen tomar tiempo $\Theta(n^2)$. Es difícil saber de antemano en qué entorno terminará ejecutándose nuestro programa. Por ello, es más seguro aleatorizar la elección del pivote. Así, nuestro QuickSort tomará tiempo *esperado* $O(n \log n)$ con cualquier input que se le dé.

6.1.3. Complejidad computacional

Los algoritmos probabilísticos y aleatorizados también irrumpen en las clases de complejidad computacional. Por ejemplo existen problemas para los que no se conoce una solución clásica de tiempo polinomial (es decir, no se sabe si están en $\mathcal{P}$) pero sí una que permite un pequeño margen de error. Uno de esos problemas es determinar si dos polinomios son iguales (por ejemplo $(x+y)(x-y) = x^2 - y^2$), que parece fácil pero puede requerir tiempo exponencial para aplicar todas las distributivas. En cambio, un algoritmo aleatorizado sencillo, al que le basta poder evaluar los polinomios sin conocer su descripción, responde en tiempo polinomial con muy baja probabilidad de error.

Una forma de entender esto es que los problemas en $\mathcal{NP}$ se resuelven con una máquina de Turing no determinística que adivina en tiempo t la hoja correcta en un árbol de todas las decisiones posibles, y luego la verifica en tiempo t'. Su tiempo total es entonces $t+t'$, que es polinomial en el tamaño del input. En cambio, la máquina determinística debe recorrer todas las hojas, que son un número exponencial con respecto a la altura t del árbol. Pero algunos problemas en $\mathcal{NP}$ tienen en realidad muchas hojas correctas, digamos una fracción $0 < p < 1$, por lo cual se pueden resolver eficientemente con un un algoritmo aleatorizado que elija una hoja al azar: éste tiene una probabilidad p de responder correctamente, por lo cual en tiempo esperado $\frac{t+t'}{p}$ encuentra una solución. Si no puede verificar la solución pero sí escoger la mejor, entonces puede repetirse k veces, con tiempo total kt, para encontrar la solución correcta con probabilidad $1-(1-p)^k$.

Algunas clases de complejidad relevantes para estos algoritmos son las siguientes:

$\mathcal{ZPP}$ son los problemas que se pueden resolver en tiempo esperado polinomial sin error (es decir, con un algoritmo tipo Las Vegas).

$\mathcal{RP}$ **y co-**$\mathcal{RP}$ son los problemas que se pueden resolver en tiempo polinomial equivocándose con probabilidad $0 < p < 1$ sólo en el caso de responder "sí", pero sin error al responder "no" (y viceversa para co-$\mathcal{RP}$). Estos son los algoritmos Monte Carlo one-sided.

$\mathcal{BPP}$ son los problemas que se pueden resolver en tiempo polinomial equivocándose con probabilidad $0 < p < \frac{1}{2}$ en caso de responder "sí" y con probabilidad $0 < p' < \frac{1}{2}$ en caso de responder "no". Estos son los algoritmos Monte Carlo two-sided.

Tenemos $\mathcal{P} \subseteq \mathcal{ZPP}$, $\mathcal{P} \subseteq \mathcal{RP} \subseteq \mathcal{NP}$ y $\mathcal{P} \subseteq$ co-$\mathcal{RP} \subseteq$ co-$\mathcal{NP}$, $\mathcal{RP} \cup$ co-$\mathcal{RP} \subseteq \mathcal{BPP}$. Asimismo, se sabe que $\mathcal{ZPP} = \mathcal{RP} \cap$ co-$\mathcal{RP}$. Actualmente se cree que $\mathcal{P} = \mathcal{BPP}$ (con lo cual

todas estas clases colapsarían y los algoritmos probabilísticos y aleatorizados no tendrían impacto en las clases de complejidad), dado que la cantidad de problemas conocidos que pueden estar en $\mathcal{BPP} - \mathcal{P}$ ha ido decreciendo. El problema mencionado de los polinomios es uno de los pocos que quedan. Sin embargo, muchas de las soluciones determinísticas encontradas son notoriamente más complicadas, y de mayor complejidad, que las probabilísticas. Veremos un caso importante a continuación.

6.2. Test de Primalidad

Determinar si un número n es primo es un problema sumamente importante, no sólo en teoría de números sino en computación, por sus aplicaciones a criptografía y hashing, por ejemplo. Si bien es fácil probar todos los posibles divisores hasta $\sqrt{n}$, un algoritmo de tiempo $O(\sqrt{n})$ es muy ineficiente para los grandes valores de n que se usan en criptografía (cientos de dígitos). En términos de complejidad, un algoritmo eficiente debe ser polinomial en función del largo ℓ del input, es decir, en los $\ell = O(\log n)$ bits usados para representar n.

Recién en 2002 se descubrió un algoritmo polinomial para determinar si n es primo o no. Sin embargo el algoritmo es complicado y, luego de varias mejoras, su complejidad es de $O(\ell^4)$ multiplicaciones, lo que aún es considerablemente grande para los valores de n que se usan en criptografía. Note que este algoritmo indica que n es compuesto sin poder dar una factorización. No se conoce un algoritmo polinomial para factorizar un número, ni siquiera de tipo probabilístico, y este desconocimiento es la base de la criptografía de clave pública.

En la práctica, para determinar si n es primo es mucho más conveniente usar el algoritmo de Miller-Rabin, que realiza $O(k \log n)$ multiplicaciones y se equivoca con probabilidad a lo sumo $\frac{1}{4^k}$, sólo en caso de responder que n es primo. Cuando responde que n es compuesto, el algoritmo no se equivoca. Es decir, es un algoritmo de tipo Monte Carlo one-sided.

El algoritmo hace lo siguiente para determinar si n es primo:

1. Sean s y d tal que $n - 1 = 2^s \cdot d$ con d impar.
2. Repetir k veces:
   - *a*) Elegir $a \in [1..n-1]$ al azar.
   - *b*) Si $a^d \not\equiv 1 \mod n$ y $\forall r \in [0..s-1]$, $a^{2^r \cdot d} \not\equiv -1 \mod n$
   - *c*) Retornar "compuesto".
3. Retornar "probablemente primo".

El algoritmo requiere $O(k \log n)$ multiplicaciones módulo n. La primera línea divide $n-1$ por 2, s veces, hasta que quede un d impar, lo cual puede requerir $O(\log n)$ iteraciones (pero puede hacerse incluso más rápido manipulando bits). En la línea 2.*b*, calcular $a^d \mod n$ requiere $O(\log d) = O(\log n)$ multiplicaciones usando un algoritmo muy simple llamado

exponenciación modular (se calcula a^1, a^2, a^4, a^8, etc. mediante ir elevando al cuadrado el valor anterior, y luego se multiplican las potencias necesarias para formar a^d). Similarmente, para calcular todos los $s = O(\log n)$ valores $a^{2^r \cdot d}$ en la línea 2.b, vamos elevando a^d al cuadrado cada vez.

Lo segundo es establecer que, si n es compuesto, entonces en cada iteración tenemos una chance de $\frac{3}{4}$ de encontrar un valor de a que satisfaga la condición de la línea 2.b. Tales valores de a se llaman *testigos* de que n es compuesto. Todo número compuesto n tiene al menos $\frac{3}{4}\,n$ testigos $a \in [0..n-1]$, por lo cual la probabilidad de que k veces no demos con uno de ellos, y respondamos incorrectamente que n es primo, es a lo sumo $\frac{1}{4^k}$.

Veamos primero que un a que cumpla con la línea 2.b es un testigo de que n es compuesto. Considere un primo n y un x tal que $x^2 \equiv 1 \mod n$. Esto es lo mismo que $(x+1)(x-1) \equiv 0 \mod n$, lo que implica $x \equiv 1 \mod n$ ó $x \equiv -1 \mod n$ (pues si un primo divide a un producto, debe dividir a uno de los factores). Es decir, las únicas raíces cuadradas de 1 módulo un primo n son 1 y -1. Sea ahora un número n con $n - 1 = 2^s \cdot d$ con d impar, como en el algoritmo. Entonces debe valer que $a^d \equiv 1 \mod n$ o que $a^{2^r \cdot d} \equiv -1 \mod n$ para algún $r \in [0..s-1]$. Esto ocurre porque $a^{2^s \cdot d} = a^{n-1} \equiv 1 \mod n$ por el Pequeño Teorema de Fermat. Si comenzamos a sacar raíces de a^{n-1} obtenemos $a^{2^{s-1} \cdot d}$, $a^{2^{s-2} \cdot d}$, ..., a^d. Como vimos, esas raíces deben ser 1 o -1. Si alguna de ellas es -1, entonces tenemos nuestro $a^{2^r \cdot d} \equiv -1 \mod n$, y si todas ellas son 1, tenemos $a^d \equiv 1 \mod n$.

El algoritmo usa la contrapositiva de esta demostración: si encontramos un a para el cual no vale ni $a^d \equiv 1 \mod n$ ni $a^{2^r \cdot d} \equiv -1 \mod n$ para ningún $r \in [0..s-1]$, entonces n no es primo. Decimos por ello que a es un testigo de que n es compuesto.

Demostrar que existen al menos $\frac{3}{4}\,n$ testigos a es algo extenso para los objetivos de este curso; daremos una referencia al final del capítulo. Por otro lado, esa es una cota bastante generosa, dado que el test es mucho más fuerte. Por ejemplo, se sabe que probando sólo $a = 2, 3, 61$ el test funciona sin error para cualquier número n de 32 bits, que probando sólo con los primeros 12 primos $a = 2, \ldots, 37$ basta para 64 bits, y que agregando $a = 41$ el test funciona sin error hasta $n < 3 \times 10^{24}$.

Generar primos. Un problema relacionado es el de generar un número primo aleatorio de ℓ bits. Una forma razonable de hacerlo es generar números al azar en ese rango y comprobar su primalidad con el test visto. Se sabe que el número de primos entre 1 y n tiende a $\frac{n}{\ln n}$, por lo que en promedio bastará con $O(\log n) = O(\ell)$ intentos para encontrar un primo, usando un algoritmo tipo Las Vegas.

6.3. Árboles Aleatorizados y Skip Lists

Veremos un par de estructuras que implementan diccionarios, es decir, mantienen un conjunto de elementos donde podemos insertar, borrar y buscar en tiempo $O(\log n)$. También podemos, en el mismo tiempo, encontrar el predecesor y el sucesor de un número dado. Ambos

algoritmos implementan todas las operaciones en tiempo *esperado* $O(\log n)$, para un conjunto de n elementos. Note la diferencia con los árboles binarios de búsqueda clásicos, que ofrecen tiempo *promedio* $O(\log n)$ bajo la suposición de que el orden en que se insertan las claves es una permutación aleatoria.

Comenzaremos con una estructura que es idéntica a un árbol binario de búsqueda clásico, pero que se encarga de producir un árbol en el que las claves se hubieran insertado en orden aleatorio, independientemente del orden en que se insertaron realmente. Luego veremos cómo ese árbol se puede usar para mantener elementos ordenados por clave y a la vez por prioridad (como en colas de prioridad). Finalmente, veremos una estructura alternativa para obtener el equivalente de la primera estructura usando menos espacio esperado.

6.3.1. Árboles aleatorizados

Nuestra tarea es mantener un árbol binario de búsqueda (ABB) para el conjunto de elementos $\{x_1, \ldots, x_n\}$ que se han insertado, aleatorizando virtualmente el orden de la entrada de modo que el árbol corresponda a cada permutación posible de esos elementos con la misma probabilidad, $\frac{1}{n!}$.

Inserción. Supongamos que, con ese invariante, tenemos en este momento un árbol binario T que corresponde a haber insertado los elementos $x_1, \ldots, x_n$, en ese orden, con el mecanismo normal de inserción de los ABBs. Si ahora se inserta un nuevo elemento x, debemos darle la misma probabilidad, $\frac{1}{n+1}$, de que se inserte en cada posible lugar dentro de la secuencia elegida $x_1, \ldots, x_n$ (no almacenaremos esta secuencia explícitamente, bastará con que cada nodo de T guarde el número de nodos en su subárbol).

En particular, decidiremos que x va al principio de la secuencia de inserción con probabilidad $\frac{1}{n+1}$. En este caso, x debe ser la raíz, como dicta el método de inserción de los ABBs. En cambio, con probabilidad $\frac{n}{n+1}$, x no va al principio de la secuencia y por lo tanto mantenemos la raíz actual (que debe ser x_1).

El segundo caso es más sencillo. Si x no está al comienzo de la secuencia de inserción, entonces x_1 sigue siendo la raíz de T, y lo era cuando llegó x a insertarse. De modo que lo que ocurrió al insertarse x fue que se lo comparó con x_1, y si $x < x_1$, lo insertamos en el hijo izquierdo de T, de otro modo lo insertamos en el hijo derecho. Es decir, continuamos como en la recursión normal de la inserción en ABBs. El código recursivo correspondiente para $insertar(T, x)$ es entonces como sigue (es fácil convertirlo a iterativo):

1. Elegir r al azar en $[1..|T| + 1]$.
2. Si $r = 1$, convertir a x en la raíz de T y retornar el resultado.
3. Si $x < T$.raíz,
   entonces T.izquierdo $\leftarrow insertar(T.\text{izquierdo}, x)$;
   si no, T.derecho $\leftarrow insertar(T.\text{derecho}, x)$.

4. Retornar T.

Lo intrigante es la línea 2. ¿Qué significa convertir a x en raíz? Significa construir el ABB T' que corresponde a la secuencia $x, x_1, \ldots, x_n$ a partir del árbol T de la secuencia $x_1, \ldots, x_n$. La raíz de T' será x, así como la de T es x_1. Llamemos T_L y T_R a los subárboles izquierdo y derecho de x en T', y T_1 y T_2 a los subárboles izquierdo y derecho de x_1 en T. Definiremos una operación $cut(T, x)$, que corta T en el par de árboles formados por los elementos menores y mayores que x, respectivamente, respetando el mismo orden de inserción que muestran en T. Es decir, $cut(T, x)$ nos entregará T_L y T_R.

Consideremos el caso $x_1 < x$. Esto significa que x_1 se convirtió en el hijo izquierdo de x, por lo tanto x_1 es la raíz de T_L. Más aún, todo el subárbol T_1 se habría comparado con x, resultando menor, y luego con x_1, resultando menor; por lo tanto se habrían organizado en el subárbol izquierdo de x_1 exactamente igual que como lo han hecho en T. Es decir, si $x_1 < x$, entonces T_L tiene a x_1 como raíz y a T_1 como su subárbol izquierdo. Los elementos de T_2, en cambio, pueden ser mayores o menores que x, de modo que podemos obtener recursivamente $(T_<, T_>) = cut(T_2, x)$. Como $T_<$ son los elementos menores que x, pero mayores que x_1, que se forman respetando el orden de inserción, $T_<$ corresponde al hijo derecho de x_1 en T_L: son los elementos llegaron en el mismo orden original, se compararon con x resultando menores, y luego con x_1 resultando mayores, y en ese orden poblaron el subárbol derecho de x_1 en T_L. Por otro lado, los elementos en $T_>$ corresponden exactamente a T_R: los elementos que se comparan con x al llegar y resultan mayores. El caso $x_1 > x$ es simétrico. El código recursivo para $cut(T, x)$ (nuevamente, fácil de convertir a iterativo) es como sigue, donde usamos la notación ⟨raíz,izquierdo,derecho⟩ para describir un árbol:

1. Si T = nulo, Retornar (nulo,nulo).
2. Si T.raíz $< x$
   - $(T_<, T_>) \leftarrow cut(T.\text{derecho}, x)$.
   - $T_L \leftarrow \langle T.\text{raíz}, T.\text{izquierdo}, T_< \rangle$.
   - $T_R \leftarrow T_>$.
3. Si no,
   - $(T_<, T_>) \leftarrow cut(T.\text{izquierdo}, x)$.
   - $T_R \leftarrow \langle T.\text{raíz}, T_>, T.\text{derecho} \rangle$.
   - $T_L \leftarrow T_<$.
4. Retornar (T_L, T_R)

Note que $cut(T, x)$ recorre el camino de la raíz de T a la hoja donde se debería insertar x en el método clásico, y por lo mismo nuestro método *insertar* también recorre el mismo camino, primero con la recursión de *insertar* y luego cambiándose a la de *cut*. Como el árbol es el correspondiente a cada uno de los órdenes posibles de inserción con la misma probabilidad, independientemente del orden real en que se insertaron los elementos, su altura *esperada* es $O(\log n)$, y ese orden es el de la inserción y el de la búsqueda (que se realiza como en un ABB clásico).

Por ejemplo, si se insertan los elementos $1, 2, 3$ en un árbol vacío con este algoritmo, se obtiene el árbol balanceado de raíz 2 con probabilidad $\frac{1}{3}$ (pues resulta de las permutaciones $2, 1, 3$ y $2, 3, 1$) , y los otros 4 árboles posibles con probabilidad $\frac{1}{6}$.

Borrado. Para borrar un determinado x_i de la secuencia $x_1, \ldots, x_n$, debemos modificar el árbol para que sea como si x_i nunca se hubiera insertado. Lo primero que hacemos es, entonces, buscar x_i en el árbol con el método normal de búsqueda, y borrarlo. Al borrarlo, queda en su lugar un nodo "vacío", del que debemos deshacernos.

Para deshacernos de la raíz vacía del subárbol de x_i, con hijos T_L y T_R, debemos considerar la subsecuencia de $x_1, \ldots, x_n$ de los nodos que cayeron en el subárbol. El primer elemento de la subsecuencia es x_i. ¡Pero con la información que almacenamos, no sabemos cuál es el segundo! Podría ser x_L, raíz de T_L, o x_R, raíz de T_R. Lo que debemos hacer es, dentro de todas las subsecuencias $x_i, \ldots$ de nodos que dan lugar a este subárbol, contar en cuántas el segundo elemento es $x_L < x_i$ y en cuántas es $x_R > x_i$. Como hay $|T_L|$ elementos del primer grupo y $|T_R|$ del segundo, la probabilidad de que luego de x_i venga x_L es $\frac{|T_L|}{|T_L|+|T_R|}$, y viceversa. En el primer caso, llamemos $T_<$ y $T_>$ a los subárboles izquierdo y derecho de x_L. El subárbol de raíz x_i, de no existir x_i, se habría inaugurado con x_L, quien sería su raíz, y su hijo izquierdo, T'_L, correspondería a $T_<$. En cambio, T'_R correspondería a los elementos de $T_>$ más los de T_R. Como los primeros son menores que los segundos, podemos definir T'_R como una raíz vacía, con subárboles izquierdo y derecho $T_>$ y T_R, respectivamente, y hemos llevado el problema del nodo vacío un nivel hacia abajo. El caso de elegir x_R es simétrico. El código recursivo para resolver la raíz vacía de un subárbol T, $merge(T)$, es como sigue:

1. Si $|T| = 1$, Retornar nulo.
2. Elegir r al azar en $[1..|T| - 1]$.
3. Si $r \leq |T.\text{izquierdo}|$
   - $I \leftarrow T.\text{izquierdo}$.
   - $T.\text{raíz} \leftarrow I.\text{raíz}$.
   - $T.\text{izquierdo} \leftarrow I.\text{izquierdo}$.
   - $T.\text{derecho} \leftarrow merge(\langle -, I.\text{derecho}, T.\text{derecho}\rangle)$.
4. Si no,

- $D \leftarrow T$.derecho.
- T.raíz $\leftarrow D$.raíz.
- T.derecho $\leftarrow D$.derecho.
- T.izquierdo $\leftarrow merge(\langle -, T.\text{izquierdo}, D.\text{izquierdo}\rangle)$.

5. Retornar T

Al realizar un recorrido desde la raíz hasta una hoja, este método de borrado tiene también un costo esperado de $O(\log n)$. El método puede agilizarse para que, en caso de que T tenga un solo hijo, simplemente borre la raíz.

6.3.2. Treaps

Un *treap* es una cruza de un árbol y una cola de prioridad (*tree* + *heap*). Almacena elementos que tienen una *clave* y una *prioridad*. Permite insertar, borrar y buscar elementos usando la clave, pero también ver y extraer el elemento con máxima prioridad.

Si bien podemos implementar un treap usando un árbol y un heap, la estructura del árbol aleatorizado nos da una implementación muy elegante, con tiempos logarítmicos bajo suposiciones razonables sobre el input (no es una estructura aleatorizada). La idea es equiparar la *prioridad* con el *momento de inserción*, de modo que una mayor prioridad corresponde a un menor tiempo de inserción. De este modo, esta implementación de treap es un ABB válido sobre las claves, pero que respeta que la prioridad del padre nunca es menor que la de sus hijos (o equivalentemente, el tiempo de inserción del padre nunca es mayor que el de sus hijos).

Para insertar en un treap con clave x y prioridad p, procedemos como en la inserción del árbol aleatorizado, sólo que en vez de usar el azar para determinar si pasamos a usar *cut*, pasamos a *cut* si la prioridad de la raíz de T es menor que p. En ese caso, convertimos x en la raíz de T y usamos $cut(T, x)$ para obtener los subárboles T_L y T_R, que serán treaps y formarán los subárboles izquierdo y derecho de x.

Para borrar x de un treap, lo buscaremos, dejaremos el nodo vacío, y usaremos un método equivalente a *merge* para deshacernos del nodo. En vez de la versión aleatorizada, este método elegirá x_L o x_R según quién tenga mayor prioridad, para llenar el nodo vaciado.

Para buscar x por clave en un treap, lo buscamos exactamente como en un ABB. Para ver el elemento de mayor prioridad, miramos la raíz. Para extraer el elemento de máxima prioridad, dejamos vacía la raíz y procedemos como en el borrado.

Los treaps permiten otras operaciones más complejas, como recorrer todos los elementos con clave entre x y x' y prioridad $\geq p$, con métodos recursivos muy sencillos.

6.3.3. Skip lists

Las *skip lists* son estructuras aleatorizadas que ofrecen garantías similares a los árboles aleatorizados. Una desventaja es que su espacio para almacenar n elementos es $O(n)$ espe-

rado, no peor caso como los árboles. Sin embargo, la constante de ese espacio esperado es menor que la de los árboles.

La skip list para una secuencia $x_1 < x_2 < \ldots < x_n$ (note que ahora vemos la secuencia en su orden numérico, no de inserción) es una lista de n *torres*, donde la i-ésima torre almacena $key(i) = x_i$ y uno o más punteros hacia torres siguientes. La altura de la torre i se define aleatoriamente al insertar x_i, mediante tirar una moneda que cae cara con probabilidad p, $k_i \geq 1$ veces hasta que salga cara. Entonces la altura de la torre es k_i (note que el valor esperado de k_i es $\frac{1}{p}$). En el piso j de la torre i, con $1 \leq j \leq k_i$, almacenamos un puntero $ptr(i, j)$ hacia la torre más cercana $i' > i$ que tenga altura $\geq j$, o nulo si no existe tal torre.

Además, la skip list guarda una torre 0, con un puntero $ptr(0, j)$ hacia la primera torre de altura $\geq j$, para $1 \leq j \leq k_{\text{máx}}$, donde $k_{\text{máx}}$ es la altura máxima de una torre.

El espacio esperado de toda la estructura es n claves más $\frac{n}{p}$ punteros. A esto se le suma la torre 0, cuya altura $k_{\text{máx}}$ es el máximo de las n torres. El valor esperado del máximo de n variables aleatorias con distribución geométrica de parámetro p es $\log_{\frac{1}{1-p}} n + O(1)$, por lo cual esto le suma sólo $O(\log n)$ al espacio.

Búsqueda. La búsqueda de un elemento x en una skip list empieza en el piso $j = k_{\text{máx}}$ de la torre $i = 0$. Si $key(ptr(i, j)) \leq x$, entonces hacemos $i \leftarrow ptr(i, j)$ (es decir, saltamos por el puntero a una torre más adelante), y si no, hacemos $j \leftarrow j - 1$ (es decir, bajamos un piso). Cuando llegamos a $j = 0$, $key(i)$ es x o su predecesor en el conjunto.

Para analizar el costo, podemos separar los dos tipos de operaciones. La cantidad de veces que se baja un piso en total es $k_{\text{máx}}$. Para ver la cantidad de saltos entre torres, consideremos la búsqueda de x_n, que es la clave más lejana. Recorreremos todas las torres de altura $k_{\text{máx}}$, y luego de la última, todas las torres siguientes de altura $k_{\text{máx}} - 1$, etc. Por simetría, esto equivale a partir de la torre 0 y recorrer por el piso 1 hasta encontrar una torre de altura ≥ 2, luego recorrer por los pisos 2 hasta encontrar una torre de altura ≥ 3, etc. En cada piso j, cada torre que visitamos se detiene en ese piso con probabilidad p, por lo que tiene más de j pisos con probabilidad $1 - p$. Por ello, recorremos en promedio $\frac{1}{1-p}$ torres hasta dar con una que tenga un piso más, y allí pasamos al piso $j+1$. De este modo, el costo esperado de todos los movimientos horizontales es $\frac{1}{1-p} k_{\text{máx}}$, y sumando los verticales tenemos $(1 + \frac{1}{1-p})k_{\text{máx}}$. El valor esperado de esto es $(1 + \frac{1}{1-p}) \log_{\frac{1}{1-p}} n$ más términos de orden inferior. Este costo es $O(\log n)$ para cualquier p fijo, y el valor óptimo de p es $\approx 0{,}72$. Con este valor de p, el costo esperado de búsqueda es $\approx 3{,}59 \ln n$ y el espacio esperado de la estructura son n claves más $\approx 1{,}39\, n$ punteros. Note que esto es inferior a los $2n$ punteros usados por el árbol aleatorizado, el cual además debe almacenar el tamaño de los subárboles (¡curiosamente, no necesitamos almacenar la altura de las torres en la skip list!). En cambio, la cantidad de comparaciones esperadas para buscar es menor en el árbol aleatorizado: $2 \ln n$.

Inserción. Para insertar x en la skip list simulamos la búsqueda, pero recordamos la última torre de altura j que visitamos, $torre(j)$, para todo j. Una vez encontrado el predecesor de

x en la torre i, creamos una nueva torre i' entre la i y la $i+1$. Determinamos su altura k aleatoriamente como explicamos, y finalmente, para cada $1 \leq j \leq k$, asignamos $ptr(i', j) \leftarrow ptr(torre(j), j)$ y $ptr(torre(j), j) \leftarrow i'$. Es decir, interponemos la torre i' entre la i y la $i+1$, interrumpiendo todos los punteros entre los pisos 1 y k para insertar el nuevo piso de la torre i'. El costo de la inserción es similar al de la búsqueda.

Borrado. Para borrar x lo buscamos como en la inserción. Al encontrarlo en la torre i, de altura k_i, antes de borrar la torre debemos conectar los punteros que llevan a la torre i con los que salen de la torre i, para todos los pisos $1 \leq j \leq k_i$: $ptr(torre(j), j) \leftarrow ptr(i, j)$. Note que podemos determinar k_i sin almacenarlo, pues llegamos a la torre i siempre desde su piso más alto. El costo de borrado es similar al de la búsqueda.

6.4. Hashing Universal y Perfecto

El *hashing* es una técnica clásica para insertar, borrar y buscar con tiempo promedio constante, lo cual no se sabe hacer en el peor caso. La idea es construir una *función de hashing* $h : \mathcal{X} \to [0..m-1]$ que mapee los objetos de su universo original $\mathcal{X}$ a una posición en una tabla de tamaño m. El elemento $x \in \mathcal{X}$ se almacena entonces en la celda $h(x)$ de la tabla, y allí mismo se busca.

Lo que hace que esta idea pueda fallar son las *colisiones*, es decir, dos elementos $x \neq y$ que son mapeados a la misma celda, $h(x) = h(y)$. Las colisiones se resuelven de distintas formas, siendo la más simple la de tener una lista enlazada en cada celda para almacenar los distintos elementos que caen en ella. De cualquier manera que se resuelvan, las colisiones atentan contra el objetivo de operar en tiempo $O(1)$. Por otro lado, es imposible evitar las colisiones si $|\mathcal{X}| > m$. Más aún, si almacenamos n elementos y $|\mathcal{X}| \geq nm$, entonces siempre, no importa cómo elijamos h, el adversario puede insertar n elementos que colisionen todos en la misma celda: basta insertar nm elementos de $\mathcal{X}$ y ver que en alguna celda caerán al menos n, entonces esos n elementos colisionan todos.

Ante la imposibilidad de tener garantías de peor caso, se recurre normalmente a una probabilística: se analizan los esquemas de hashing como si $h(x)$ fuera una *variable aleatoria* distribuida uniformemente en $[0..m-1]$. De este modo, la cantidad esperada de elementos por celda será $\frac{n}{m}$, lo que será $O(1)$ si hacemos que el tamaño de la tabla sea proporcional al número de elementos que se almacenarán. Si no conocemos n de antemano, podemos hacer que la tabla se vaya duplicando cuando $\frac{n}{m}$ exceda un cierto valor permitido, y volviendo a insertar todos los elementos en la nueva tabla. Esto agrega un tiempo amortizado constante por operación, como vimos en el capítulo de análisis amortizado.

Sin embargo, h debe ser determinística, o no encontraremos un elemento x que insertamos antes. Se busca que h "se comporte" como si fuera aleatoria. Existen algunos tipos de funciones de hashing conocidos por "distribuir bien" los valores, "destruir posibles regularidades del input", etc. Pero mientras su elección sea determinística, la función h distribuirá

bien los valores sólo si estos tienen una cierta distribución, es decir, el comportamiento del hashing será promedio, no esperado, y dependerá del input. Habrá inputs que harán fallar sistemáticamente a la función de hashing elegida.

En esta sección veremos una forma aleatorizada de elegir funciones de hashing, que garanticen el comportamiento esperado deseado, independientemente de qué valores se inserten en la tabla. Note que, de todos modos, una vez elegida h, se debe seguirla usando durante el tiempo de vida de la estructura de datos. Sin embargo, una secuencia de inserciones que haga fallar h no lo logrará sistemáticamente: repetir la secuencia con otra elección de h probablemente hará que la segunda vez no falle.

6.4.1. Hashing universal

Supongamos que nuestro universo es $\mathcal{X} = [0..N-1]$, con $N > m$. Diremos que $\mathcal{H}$ es una *familia universal* de funciones de hashing de $[0..N-1]$ en $[0..m-1]$ si, para todo $x \neq y$,

$$Pr(h(x) = h(y)) \ \leq \ \frac{1}{m},$$

donde la probabilidad se toma sobre las posibles elecciones de funciones $h \in \mathcal{H}$ (eligiendo uniformemente).

Si tenemos una familia universal $\mathcal{H}$, todo lo que tenemos que hacer es elegir un h aleatoriamente de la familia cada vez que creamos una tabla de hashing. Para ver que la propiedad de universalidad es suficientemente buena, definamos S como el conjunto de claves que vamos a almacenar, $n = |S|$, y la variable aleatoria

$$C_{x,y} \ = \ 1 \text{ si } h(x) = h(y) \text{ y } 0 \text{ si no.}$$

Las $C_{x,y}$ son variables aleatorias, pues dependen de la elección al azar de h. Si h se elige uniformemente de una familia universal $\mathcal{H}$, tenemos por definición

$$Pr(C_{x,y} = 1) \ \leq \ \frac{1}{m}.$$

Definamos también

$$C_{x,S} \ = \ \text{la cantidad de elementos de } S \text{ que colisionan con } x.$$

Dicho de otro modo,

$$C_{x,S} \ = \ \sum_{y \in S} C_{x,y}.$$

También podemos ver $C_{x,S}$ como el largo de la lista enlazada en la celda $h(x)$, lo cual es proporcional al costo de buscar, insertar o borrar x. El costo esperado de estas operaciones es entonces

$$\mathbb{E}(C_{x,S}) = \ \mathbb{E}\left(\sum_{y \in S} C_{x,y}\right) = \ \mathbb{E}(C_{x,x}) + \sum_{y \in S - \{x\}} \mathbb{E}(C_{x,y}) \ = \ 1 + \sum_{y \in S - \{x\}} Pr(C_{x,y} = 1) \ \leq \ 1 + \frac{n-1}{m}.$$

Es decir, el costo esperado de las operaciones será $O(1)$ si $n = O(m)$.

Una familia universal. Existen varias familias universales conocidas. Una sencilla es la siguiente: dado un primo $p \geq N$, definimos

$$\mathcal{H} \quad = \quad \{h_{a,b},\ a \in [1..p-1] \text{ y } b \in [0..p-1]\}$$

donde

$$h_{a,b}(x) = ((ax+b) \mod p) \mod m.$$

Para ver que esta familia es universal, consideremos cuatro números $r, s, x, y \in [0..p-1]$, con $r \neq s$ y $x \neq y$, y calculemos

$$Pr((ax+b \equiv r \mod p) \text{ y } (ay+b \equiv s \mod p)).$$

Si esto ocurre, entonces $a(x-y) \equiv r-s \mod p$. Siendo p primo, y por lo tanto primo relativo con $x-y$ (pues $x, y < N \leq p$, con lo cual $x-y$ no puede ser múltiplo de p), la ecuación tiene una única solución, $a \equiv (x-y)^{-1}(r-s) \mod p$. Y dado este valor de a, existe un único valor $b \equiv r - ax \mod p$ (o equivalentemente $b \equiv s - ay \mod p$). Es decir, de las $|\mathcal{H}| = p(p-1)$ posibles funciones $h_{a,b}$ para elegir, existe exactamente una con el valor adecuado de a y b para los r, s, x, y que elegimos. Entonces la probabilidad que buscamos es

$$Pr((ax+b \equiv r \mod p) \text{ y } (ay+b \equiv s \mod p)) \quad = \quad \frac{1}{p(p-1)}.$$

Note, en cambio, que esto no puede ocurrir si $r = s$ y $x \neq y$.

Para que x e y colisionen, es decir $h(x) = h(y)$, debe ocurrir lo anterior para algún par $r \equiv s \mod m$ con $r \neq s$, es decir, $s = r \pm km$ para $k > 0$. Tenemos p formas de elegir r. Y para cada elección de r, tenemos a lo más $\lceil \frac{p}{m} \rceil - 1$ formas de elegir s. En total, el número de pares $r \neq s$ que generan una colisión es

$$p \cdot \left(\left\lceil \frac{p}{m} \right\rceil - 1 \right) \quad = \quad p \cdot \left(\left\lfloor \frac{p+m-1}{m} \right\rfloor - 1 \right) \quad = \quad p \cdot \left\lfloor \frac{p-1}{m} \right\rfloor \quad \leq \quad \frac{p(p-1)}{m}.$$

La probabilidad, al elegir h, de que cada uno de esos $\frac{p(p-1)}{m}$ pares $r \neq s$ genere una colisión módulo p es $\frac{1}{p(p-1)}$, con lo cual la probabilidad de una colisión es a lo sumo $\frac{1}{m}$.

Utilizar esta familia con una tabla de tamaño m es muy sencillo: generamos un primo $p \geq N$ con el algoritmo ya discutido, y luego generamos a y b al azar. Luego usamos $h(x) = ((ax+b) \mod p) \mod m$ como nuestra función de hash, que mapea los elementos de $[0..N-1]$ a $[0..m-1]$.

¿Es necesaria la b? Podemos preguntarnos si es necesario incluir b en la fórmula, o podríamos tener simplemente una familia $h_a(x) = (ax \mod p) \mod m$. En ese caso tendríamos $|\mathcal{H}| = p-1$ y la colisión antes de tomar módulo m implicaría $a \equiv (x-y)^{-1}(r-s) \mod p$. Entonces la probabilidad de colisión es a lo más $\frac{1}{|\mathcal{H}|} = \frac{1}{p-1}$. Ahora bien, la condición sobre

a es necesaria pero puede no ser suficiente, pues requerimos que ese a satisfaga $r - ax \equiv 0$ mód p (antes elegíamos b precisamente para que $r - ax \equiv b$). Sustituyendo el valor de a, esto equivale a $sx \equiv ry$ mód p. Así, podemos elegir r de p formas pero entonces debe ser $s \equiv x^{-1}yr$ mód p y m debe dividir a $s - r = (x^{-1}y - 1)r$. Esto puede ser factible para los p valores de r si m divide a $x^{-1}y - 1$, o sólo para $\frac{p}{m}$ de ellos si m y $x^{-1}y - 1$ son primos relativos. Es decir, la probabilidad de colisión depende de x e y, pudiendo variar desde tanto como $\frac{p}{p-1}$ (una cota mayor a 1) hasta tan poco como $\frac{p}{m(p-1)}$ (cercana a la que obtuvimos; algunos de estos cálculos tienen excepciones si hay valores cero). El esquema que vimos, en cambio, ofrece una probabilidad de colisión de $\frac{1}{m}$ independientemente del input.

Otros universos. Cuando el universo $\mathcal{X}$ no es de la forma $[0..N-1]$, debemos llevarlo a esa forma. Por ejemplo, los strings sobre el alfabeto $[0..\sigma-1]$ pueden interpretarse como números en base σ, es decir, $x_1 \dots x_\ell = \left(\sum_{i=1}^{\ell} x_i\, \sigma^{i-1}\right)$ mód p. Esto es popular, pero es determinístico y un adversario puede elegir strings en concreto que dan el mismo valor módulo p para producirnos colisiones. Una versión aleatorizada que nos da una familia universal es

$$h(x_1 \dots x_\ell) \quad = \quad \left(\left(b + \sum_{i=1}^{\ell} a_i x_i\right) \text{ mód } p\right) \text{ mód } m,$$

donde $b \in [0..p-1]$ y los $a_i \in [1..p-1]$ se eligen al azar (necesitamos tantos valores a_i como el string más largo; podemos ir generando nuevos al irlos necesitando).

Funciones más rápidas. Si el tamaño m de nuestra tabla es una potencia de 2, $m = 2^\ell$, entonces calcular el módulo m es barato, pues equivale a quedarse con los ℓ bits más bajos del valor. En cambio, calcular módulo p para un p primo es una operación relativamente lenta. Una buena opción es elegir p como un primo de Mersenne, que tiene la forma $2^k - 1$, de modo que el módulo se puede reducir a bitwise-ands y shifts: $x \equiv (x \text{ mód } 2^k) + (x \div 2^k) \text{ mód } p$, donde $\div$ es la parte entera de la división. Así, x mód $2^k = x \,\&\, ((1 \ll k) - 1)$, $x \div 2^k = x \gg k$, y el último mód p sólo requiere restar p si el valor es $\geq p$, pues éste es menor a $2p$. Los primos de Mersenne son bastante escasos, pero hay buenas opciones para enteros de 32 a 128 bits (por ejemplo, $2^{31} - 1$, $2^{61} - 1$, $2^{89} - 1$ y $2^{127} - 1$ lo son).

Otra familia universal que usa sólo módulos y divisiones por potencias de 2 es

$$h_{a,b}(x) \quad = \quad ((ax + b) \text{ mód } 2^k) \div 2^\ell,$$

con $k \geq \ell$. Esta función mapea x a $[0..2^{k-\ell} - 1]$.

6.4.2. Hashing perfecto

En algunos casos, conocemos de antemano los n elementos S que insertaremos en la tabla, y sólo nos interesa buscarlos más adelante. En este caso podríamos pensar en generar

una función de hashing h que no produjera ninguna colisión para los elementos de S. A este tipo de función de hashing se la llama *perfecta*. Al no generar colisiones, no necesitamos almacenar listas enlazadas ni verificar si es efectivamente la clave x quien está almacenada en la celda $h(x)$ (aunque si buscamos elementos $x \notin S$ aún necesitaremos comparar las claves para verificar que no sea una colisión fuera de S).

Es tentador generar funciones de una familia universal $\mathcal{H}$ hasta dar con una perfecta, pero ¿es eficiente? ¿obtendremos una función perfecta con una cantidad razonable de intentos?

Veamos cuál es la probabilidad de que una función $h \in \mathcal{H}$ sea perfecta. La probabilidad de que $x \neq y$ colisionen es $\leq \frac{1}{m}$. La probabilidad de que *alguno* de los $\frac{n(n-1)}{2}$ pares colisione es a lo más $\frac{1}{m} \cdot \frac{n(n-1)}{2} < \frac{1}{2} \cdot \frac{n^2}{m}$. Por lo tanto, si elegimos una tabla de tamaño $m = n^2$, tenemos una chance de $\frac{1}{2}$ de que una función $h \in \mathcal{H}$ elegida al azar sea perfecta. Esto nos da un algoritmo tipo Las Vegas para encontrar una función perfecta. El número esperado de intentos es 2, y en cada intento podemos verificar en tiempo $O(n)$ si hay alguna colisión (mediante insertar todos los elementos de S en la tabla). El tiempo esperado de este algoritmo es entonces $O(n)$.

El problema es que este algoritmo requiere una tabla demasiado grande para tener una chance aceptable de encontrar una función perfecta. Querríamos una tabla de tamaño $O(n)$, y construirla en tiempo esperado $O(n)$ también.

Una solución es un esquema de dos niveles. Primero elegiremos una función de hashing *distribuidora* h, de S a $[0..n-1]$, aleatoriamente de una familia universal. Esta función difícilmente será perfecta, pero esperamos que distribuya los elementos de S más o menos uniformemente. Luego tendremos una tabla de hashing perfecto asociada a cada celda de la tabla principal, para almacenar los elementos que h envió a esa celda.

Concretamente, sea B_i el conjunto de elementos de S que h pone en la celda $0 \leq i < n$, y $b_i = |B_i|$. Si ahora creáramos una función de hashing perfecto h_i para almacenar los elementos de cada B_i en una tabla propia de tamaño b_i^2, entonces fácilmente encontraríamos una función perfecta en tiempo $O(b_i)$. El tiempo total para encontrar las n funciones h_i sería $O(\sum b_i) = O(n)$. El espacio, en cambio, sería $X = \sum b_i^2$, una variable aleatoria.

Veamos el valor esperado de X. Para ello, notemos que

$$\sum_{0 \leq i < n} b_i^2 \;=\; \sum_{x,y \in S} C_{x,y},$$

pues si $B_i = \{x_1, \ldots, x_{b_i}\}$, cada par (x_i, x_j) aporta un $C_{x_i,x_j} = 1$. La esperanza es entonces

$$\mathbb{E}(X) \;=\; \mathbb{E}\left(\sum_{x,y \in S} C_{x,y}\right) \;=\; \sum_{x \in S} \mathbb{E}(C_{x,x}) + \sum_{x \neq y \in S} \mathbb{E}(C_{x,y}) \;\leq\; n + n(n-1) \cdot \frac{1}{n} \;<\; 2n,$$

donde en la primera desigualdad usamos que $\mathbb{E}(C_{x,y}) = Pr(C_{x,y} = 1) \leq \frac{1}{n}$ porque h es universal en $[0..n-1]$.

Tenemos entonces que la esperanza de la variable aleatoria $X = \sum b_i^2$ es menos de $2n$. Como X es el espacio de todas las tablas del segundo nivel, si $X < 2n$ entonces nuestro

espacio total es $O(n)$, incluyendo la tabla para h y los valores a_i y b_i almacenados para h_i. Quisiéramos entonces elegir funciones $h \in \mathcal{H}$ al azar hasta obtener una donde X fuera efectivamente $2n$, o al menos $O(n)$. ¿Cuántos intentos debemos hacer? La desigualdad de Markov nos dice que si $\mu = \mathbb{E}(X)$, entonces

$$Pr(X \ \geq \ k\mu) \ \leq \ \frac{1}{k},$$

por lo tanto, como $\mu < 2n$, $Pr(X \geq 4n) \leq \frac{1}{2}$. Es decir, tenemos una chance de al menos $\frac{1}{2}$ de que, al elegir h, nos produzca un espacio total de $\sum b_i^2 \leq 4n$.

El algoritmo tipo Las Vegas es, entonces, como sigue. Elegimos una función distribuidora h de S en $[0..n-1]$, mapeamos todos sus elementos en una tabla de contadores, donde iremos acumulando las cantidades b_i de elementos que caen en cada celda. Luego, verificaremos si $\sum b_i^2 \leq 4n$. Si no es el caso, es que elegimos una h mala, que nos requerirá mucho espacio, por lo cual volvemos a probar con otra h. El número esperado de intentos es 2, por lo que el tiempo esperado hasta que encontremos una buena función distribuidora h es $O(n)$. Una vez encontrada, construimos los conjuntos B_i y para cada uno creamos una función de hashing perfecto h_i en tiempo $O(b_i)$ y espacio b_i^2. Esta función (su a_i y b_i) se almacena en la i-ésima celda de la tabla. El tiempo esperado de construcción, y el espacio total, son $O(n)$.

Para buscar x, primero vamos a la celda $i = h(x)$. Allí tenemos guardada la función h_i, con la que mapearemos x a la posición $h_i(x)$ de la tabla asociada a la celda i, donde se guardan los elementos de B_i. El tiempo es $O(1)$ en el peor caso.

6.5. Ficha Resumen

- Conceptos de algoritmos probabilísticos y aleatorizados, tiempo esperado.
- Verificar si n es primo: $O(k \log n)$ multiplicaciones y se puede equivocar con probabilidad $\leq \frac{1}{4^k}$ al decir que es primo.
- Generar un primo entre n y $2n$: $O(k \log^2 n)$ multiplicaciones esperadas, puede entregar un no-primo con probabilidad $\leq \frac{1}{4^k}$.
- Árboles aleatorizados: tiempo $O(\log n)$ esperado para insertar, borrar y buscar.
- Skip lists: similar a árboles aleatorizados. El espacio es $O(n)$ esperado, no peor caso, aunque con una mejor constante esperada (y peor constante de tiempo esperado).
- Hashing universal: tiempo $O(1)$ esperado para todas las operaciones con espacio $O(n)$ eligiendo funciones de hash en forma aleatorizada.
- Hashing perfecto: tiempo esperado de construcción $O(n)$, espacio $O(n)$, tiempo de búsqueda $O(1)$ en el peor caso.

6.6. Material Suplementario

Cormen et al. [CLRS01, cap. 5] dedican un capítulo a conceptos básicos de algoritmos aleatorizados. Más adelante en el libro analizan versiones aleatorizadas de QuickSort [CLRS01, sec. 7.3 y 7.4] y QuickSelect [CLRS01, sec. 9.2]. Brassard y Bratley [BB88, cap. 8] dedican un capítulo a algoritmos probabilísticos y aleatorizados. Explican la clasificación en Monte Carlo y Las Vegas, y le llaman tipo "Sherwood" a los aleatorizados que no son probabilísticos (es decir, que no se equivocan y siempre terminan, como nuestras estructuras de datos aleatorizadas). Dan un buen número de ejemplos pequeños y medianos de algoritmos de cada tipo, referentes a problemas de cálculo numérico, de aritmética entera (incluyendo el test de primalidad que vimos), de problemas combinatoriales, y algunos de estructuras de datos aleatorizadas de menor importancia. Manber [Man89, sec. 6.9] le dedica una sección a algoritmos probabilísticos y aleatorizados. Discute conceptos básicos y los tipos Monte Carlo y Las Vegas, presenta un generador seudoaleatorio popular, y un ejemplo simple. También discute una técnica para convertir algoritmos aleatorizados tipo Las Vegas en determinísticos. Kleinberg y Tardos [KT06, cap. 13] dedican un muy buen capítulo a estos algoritmos. Esto incluye conceptos básicos de probabilidades, una descripción mucho más sofisticada de nuestro ejemplo de Ethernet como un problema genérico de acceso a recursos, los algoritmos QuickSort y QuickSelect aleatorizados, y otros que mencionaremos más adelante. El libro de Motwani y Raghavan [MR95] se dedica completamente a estos algoritmos y describe un número importante de técnicas de diseño junto con algoritmos y estructuras de datos concretos. En el primer capítulo incluye una descripción de las clases de complejidad relacionadas con la aleatorización. También incluye [MR95, sec. 14.6] algoritmos para verificar primalidad del estilo del que vimos, demostrando su probabilidad de error.

Motwani y Raghavan [MR95, sec. 8.1 y 8.2] describen los treaps (los cuales no son aleatorizados) y luego muestran cómo los árboles binarios aleatorizados se obtienen insertando los elementos en un treap con una prioridad asignada en forma aleatoria. Sin embargo, usa rotaciones en vez de *cut* y *merge* para insertar y borrar. Asimismo, en la sección 8.3, describen las skip lists.

Cormen et al. [CLRS01, sec. 11.5] describen la construcción del hashing perfecto como la vimos en el capítulo; en los ejercicios describen la función de hashing universal. Mehlhorn y Sanders [MS08, sec. 4.2] dedican una excelente sección al hashing universal, donde describen una cantidad de familias universales (entre ellas la que vimos en el capítulo), e incluso una forma eficiente de encontrar un primo mayor a N y cercano. Usan el término "c-universal" en una forma no ortodoxa, significando que la probabilidad de colisión es $\leq \frac{c}{n}$. Más adelante [MS08, sec. 4.5] presentan el hashing perfecto en forma muy parecida a la del capítulo, aunque describen brevemente cómo se podrían permitir inserciones y borrados. Kleinberg y Tardos [KT06, sec. 13.6] también explican de buena forma el hashing universal y perfecto, usando otra familia universal (descrita también por Mehlhorn y Sanders). Motwani y Raghavan [MR95, sec. 8.4] presentan el hashing universal, usando el término equivalente de "2-universal" (el 2 viene de que son pares x e y) e introduciendo los conceptos de

"strongly 2-universal" y su generalización "strongly k-universal" (que significa que k variables aleatorias $h(x_i)$ cualesquiera son independientes; la familia que vimos es realmente strongly 2-universal). Presentan la misma familia $\mathcal{H}$ que usamos en el capítulo, e incluyen una detallada presentación, algo distinta de la nuestra, del hashing perfecto. Navarro [Nav16, sec. 4.5.3] describe una construcción de hashing perfecto más sofisticada, que requiere menos espacio de almacenamiento.

Lee et al. [LTCT05, cap. 11] dedican un capítulo a algoritmos aleatorizados y probabilísticos. Si bien la parte de conceptos básicos es débil, incluyen varios problemas interesantes de mediana complejidad, como encontrar el par de puntos más cercanos entre n en tiempo esperado $O(n)$ usando Las Vegas (los algoritmos determinísticos son $O(n \log n)$), una variante del algoritmo de Miller-Rabin para verificar primalidad, búsqueda en texto usando Monte Carlo, y un algoritmo Las Vegas de tiempo esperado $O(n+e)$ para encontrar el árbol cobertor mínimo en un grafo de n nodos y e aristas. Kleinberg y Tardos [KT06, cap. 13] también describen problemas de cierta complejidad, como encontrar el corte mínimo de un grafo con un algoritmo Monte Carlo, algoritmos en línea y aproximados aleatorizados (donde se puede conseguir, por ejemplo, ser $O(\log k)$ competitivo en promedio para el paginado), nuevamente el par de puntos más cercano, y varios problemas de algoritmos distribuidos que la aleatorización simplifica notablemente. El libro de Motwani y Raghavan [MR95] es sin duda la referencia más completa para algoritmos probabilísticos y aleatorizados, incluyendo problemas de geometría, grafos y teoría de números, y algoritmos aproximados, paralelos y en línea.

Otras fuentes online de interés:

- `jeffe.cs.illinois.edu/teaching/algorithms/notes/09-nutsbolts.pdf`
- `www.cs.cornell.edu/courses/cs4820/2010sp/handouts/MillerRabin.pdf`
- `jeffe.cs.illinois.edu/teaching/algorithms/notes/10-treaps.pdf`
- `www.cs.cmu.edu/afs/cs/academic/class/15210-s15/www/lectures/bst-notes.pdf`
- `www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/13RandomizedAlgorithms.pdf`
- `www.cse.iitk.ac.in/users/sbaswana/randomized-algo.html`
- `www.cs.ubc.ca/~nickhar/W12`
- `www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf`
- `www.youtube.com/watch?v=2g9OSRKJuzM`
- `www.youtube.com/watch?v=z0lJ2k0sl1g`

Capítulo 7

Algoritmos Aproximados

Se llaman problemas de *decisión* a aquellos en los que debe responderse sí o no frente a un input dado. Los problemas de decisión *NP-completos* son aquellos que, hasta donde sabemos, no se pueden resolver en tiempo polinomial en computadores realistas, sino que requieren de una Máquina de Turing no determinística, la cual "adivina" la respuesta correcta y luego sólo debemos verificar que lo es. Dicho de otro modo, las únicas soluciones conocidas (y probablemente las únicas existentes) para problemas NP-completos en un computador realista requieren tiempo exponencial en el tamaño de la entrada.

Los problemas de *optimización* son aquellos en los cuales hay que construir un objeto que maximiza o minimiza una determinada función. Por ejemplo, dado un grafo, encontrar un subgrafo maximal que sea un clique (es decir, que todos los nodos estén conectados con todos). Los problemas de optimización suelen estar relacionados con problemas de decisión. Por ejemplo, el problema de decisión CLIQUE es, dado un grafo G y un número k, determinar si G tiene o no un clique de tamaño k. Es fácil ver que, si podemos resolver el problema de decisión en tiempo polinomial T, también podemos resolver el problema de optimización en tiempo $O(T \cdot \log |G|)$ mediante búsqueda binaria, el cual también es polinomial en el tamaño del input (que es $|G|$). A la inversa, es trivial resolver el problema de decisión si tenemos una solución al problema de optimización.

Por ello, podemos hablar en general de problemas de optimización NP-completos, es decir, para los cuales no tenemos esperanza de encontrar una solución de tiempo polinomial. Los algoritmos aproximados ofrecen una salida que puede ser útil en muchos casos reales: encuentran en tiempo polinomial una solución cuya "distancia" multiplicativa a la solución óptima puede garantizarse. Más formalmente, un algoritmo A para resolver un problema de maximización es una $\rho(n)$-aproximación si

$$\forall n, \quad \max_{I,|I|=n} \frac{S_{OPT}(I)}{S_A(I)} \ \leq \ \rho(n),$$

donde $S_{OPT}(I)$ es el valor de la solución óptima para el input I, y $S_A(I)$ es el valor de la solución que entrega el algoritmo A para el input I. Similarmente, para un algoritmo de

minimización debe cumplirse

$$\forall n, \quad \max_{I,|I|=n} \frac{S_A(I)}{S_{OPT}(I)} \ \leq \ \rho(n).$$

Veremos que, con esta óptica, no todos problemas NP-completos (en su versión de optimización) son igualmente intratables. Algunos resultan ser aproximables por una constante, por ejemplo $\rho(n) = 2$, mientras que otros son aproximables por una función que empeora con n, por ejemplo $\rho(n) = \log n$. También veremos que otros problemas no son aproximables en absoluto. Finalmente, veremos que algunos problemas admiten lo que se llama un *esquema de aproximación polinomial*, en el cual se le entrega un input adicional ϵ, tan pequeño como se desee, y el algoritmo produce una $(1+\epsilon)$-aproximación. El costo del algoritmo debe ser polinomial en n, pero podría ser exponencial en $\frac{1}{\epsilon}$, por ejemplo podría ser $O(n^{2/\epsilon})$. Mejor aún es un *esquema de aproximación completamente polinomial*, en el cual el costo también debe ser polinomial en $\frac{1}{\epsilon}$, por ejemplo $O(\frac{n^2}{\epsilon})$.

7.1. Recubrimiento de Vértices

Un *recubrimiento de vértices* de un grafo $G(V, E)$ es un subconjunto de nodos tal que toda arista incide en algún nodo elegido. Es decir, es un $V' \subseteq V$, tal que para toda arista $(u, v) \in E$, se tiene que $u \in V'$ ó $v \in V'$ (o ambos). El problema de decisión de decir si G tiene un recubrimiento V' de determinado tamaño k, es NP-completo. En el problema de optimización, se desea encontrar un recubrimiento de vértices de tamaño mínimo.

7.1.1. Vértices sin Pesos

En la versión simple del problema, los vértices son todos equivalentes, y entonces se desea minimizar $|V'|$. Podemos mostrar fácilmente cómo construir una 2-aproximación a este problema. Comencemos con $V' = \emptyset$ y vayamos extrayendo una a una las aristas de E. Por cada (u, v) que extraigamos, agreguemos $\{u, v\}$ a V'. Esta arista estará entonces cubierta, pues ambos extremos están en V' (bastaría con uno solo). Incluso, cualquier otra arista de E que incida en los nodos u ó v también estará cubierta, por lo cual también las eliminamos de E. Después volvemos a sacar una arista de las que restan en E, metemos ambos nodos en V', y repetimos el proceso hasta que E sea vacío.

Está claro que el conjunto V' resultante es un recubrimiento de vértices, pues toda arista que se sacó de E estaba cubierta por algún nodo de V'. Para ver que además su tamaño no puede ser más que el doble del óptimo, notemos que, cuando metemos u y v en V' la primera vez, alguno de los dos tiene que estar en cualquier recubrimiento óptimo, pues si no la arista (u, v) no estaría cubierta. Es decir, por cada dos nodos que metemos en V', el algoritmo óptimo tiene que haber metido al menos uno. Dado que luego se eliminan las aristas que inciden en u ó v, todos los pares (u, v) cuyos extremos incluimos en V' son de

nodos disjuntos, por lo cual el argumento aplica individualmente a cada par: de cada par, el algoritmo óptimo debe meter un nodo distinto en su solución. Se sigue que nuestro algoritmo es una 2-aproximación. Por supuesto, es de tiempo polinomial.

7.1.2. Vértices con Pesos

Una variante más sofisticada le asigna costos $c(v)$ a los nodos $v \in V$, y se busca un V' que minimice $\sum_{v \in V'} c(v)$. Nuevamente, es posible obtener una 2-aproximación, pero la solución es bastante más sofisticada.

Comencemos por expresar este problema de minimización como un problema de programación entera. Esta es una técnica muy fructífera para encontrar buenas aproximaciones. Tendremos una variable $x(v)$ para cada $v \in V$, donde $x(v) = 0$ indicará que $v \notin V'$ y $x(v) = 1$ que $v \in V'$. Lo que queremos minimizar es entonces $\sum_{v \in V} c(v) \cdot x(v)$. La restricción de que toda arista esté cubierta la expresaremos como $x(u) + x(v) \geq 1$ para todo $(u, v) \in E$. El problema resultante se ve entonces así:

$$\begin{aligned} \text{Minimizar} \quad & \sum_{v \in V} c(v) \cdot x(v) \\ \text{Sujeto a} \quad & \\ & x(u) + x(v) \geq 1, \quad \forall (u, v) \in E \\ & x(v) \in \{0, 1\}, \quad \forall v \in V \end{aligned}$$

La última condición hace que este sea un problema de programación entera y no lineal. Lamentablemente la optimización entera también es NP-completa, por lo cual no parece que hayamos avanzado mucho. Sin embargo, si cambiamos la última restricción por

$$0 \leq x(v) \leq 1, \quad \forall v \in V,$$

tendremos un problema de programación lineal, que sí se puede resolver en tiempo polinomial.

El problema, claro, es que la solución a este problema le asignará a cada $v \in V$ un valor real $x(v) \in [0, 1]$, lo cual no podemos usar directamente para modelar la solución a nuestro problema original de recubrimiento de vértices. Lo que haremos será algo intuitivamente razonable: diremos que $v \in V'$ sii $x(v) \geq 0{,}5$.

Veamos primero que esto entrega un recubrimiento de vértices. Para cada $(u, v) \in E$, debe cumplirse que $x(u) + x(v) \geq 1$, por lo cual alguna de las dos debe ser $\geq 0{,}5$, y entonces será incluida en V'. Por lo tanto, nuestro procedimiento entrega una solución válida.

Segundo, veamos que la solución es una 2-aproximación. Para ello, observemos que la solución al problema de programación lineal no puede ser más costosa que la de programación entera, dado que permite un superconjunto de soluciones. Y ahora, observemos que nuestra solución cuesta a lo sumo el doble que la de programación lineal. En efecto, lo que hacemos es equivalente a que, si $x(v) < 0{,}5$, entonces asignamos $x(v) \leftarrow 0$ (lo cual reduce el costo), y que, si $x(v) \geq 0{,}5$, entonces asignamos $x(v) \leftarrow 1$ (lo cual a lo sumo duplica el costo).

7.2. El Viajante de Comercio

Otro famoso problema NP-completo es el de determinar si un grafo dirigido $G(V, E)$ tiene un *circuito hamiltoniano*. Este es un camino que pasa por cada nodo exactamente una vez y luego vuelve al nodo original. En la versión de optimización, el grafo G es completo (es decir, $E = V \times V$) pero las aristas tienen un costo no negativo $c(u, v)$. Se busca entonces un circuito hamiltoniano que minimice la suma de los costos de sus aristas. El nombre del problema alude a un viajante que deba visitar cada ciudad (nodo) exactamente una vez y volver a la suya.

7.2.1. Caso General

Veremos primero que este problema es inaproximable. La forma de demostrar esto es ver que, si existiera una ρ-aproximación para el problema del viajante de comercio, entonces podríamos resolver el problema del circuito hamiltoniano en tiempo polinomial. Lograr esto con cualquier problema NP-completo implica inmediatamente que se puede lograr para todos, lo cual se considera altamente improbable (y el concepto de aproximación pierde sentido).

Supongamos que existe tal ρ-aproximación. Consideremos un problema de circuito hamiltoniano $G(V, E)$. A partir de él, diseñaremos el siguiente problema de viajante de comercio: el grafo $G'(V', E')$ es el grafo completo donde $V' = V$, y los costos de las aristas se definen de la siguiente manera: si $(u, v) \in E$ (en el grafo original), entonces $c(u, v) = 1$ (en el grafo del viajante de comercio); si no, entonces $c(u, v) = \rho \cdot |V| + 1$.

La intuición es que hacemos que pasar por las aristas que existen en G sea muy barato (costo 1), mientras que pasar por las que no existen es muy caro (costo $\rho \cdot |V| + 1$). La diferencia es tanta, que incluso una ρ-aproximación es capaz de distinguir si existe o no un circuito que use solamente las aristas permitidas (las de E).

Concretamente, note que todo circuito tiene exactamente $|V|$ aristas. Si existe un circuito hamiltoniano en G, entonces ese circuito tiene costo total $|V|$ en G' (pues todas las aristas son de costo 1). Esto significa que una ρ-aproximación entregará una solución de costo a lo sumo $\rho \cdot |V|$. En cambio, si no existe un circuito hamiltoniano en G, entonces todo circuito en G' debe usar al menos una arista que no está en E, la cual tiene costo $\rho \cdot |V| + 1$. Por lo tanto el costo de incluso la mejor solución es mayor que $\rho \cdot |V|$. Esto significa que, viendo el costo que obtiene una ρ-aproximación, que corre en tiempo polinomial, somos capaces de decir si G tiene o no un circuito hamiltoniano, y hemos resuelto un problema NP-completo en tiempo polinomial.

7.2.2. Costos Métricos

En cambio, es posible obtener una 2-aproximación para este problema en el caso particular (pero plausible) de que los costos en E satisfagan los axiomas de una métrica: $c(u, u) = 0$

(reflexividad), $c(u,v) = c(v,u)$ (simetría), y $c(u,v) + c(v,w) \geq c(u,w)$ (desigualdad triangular). Incluso existe una 1,5-aproximación (ver las referencias al final del capítulo).

Podemos obtener una 2-aproximación de la siguiente forma. Comencemos generando un árbol cobertor mínimo T de G. El costo $c(T)$ de T (es decir, la suma de los costos de sus aristas) tiene que ser menor que el de cualquier circuito hamiltoniano, pues si al circuito se le saca una arista el resultado es un camino que toca todos los nodos, y eso también es un árbol cobertor de G. Por lo tanto, si c^* es el costo óptimo del viajante de comercio, tenemos que $c(T) \leq c^*$. Note que no importa la dirección de las aristas para calcular su costo porque estamos suponiendo simetría, $c(u,v) = c(v,u)$.

Claro que T no es el circuito que necesitamos, sino un árbol. Obtendremos un circuito C a partir de T mediante recorrerlo en orden DFS partiendo de cualquiera de sus nodos. Como cada arista de T se recorre dos veces en C (una al ir y otra al volver del DFS), tenemos que $c(C) = 2 \cdot c(T)$ (nuevamente estamos usando la simetría de los costos).

Si bien C es un circuito que incluye a todos los nodos de G, resulta que puede pasar varias veces por un mismo nodo (todos los nodos internos del recorrido DFS), por lo cual no es aún un circuito hamiltoniano. Para convertirlo en uno, escribiremos la lista de los nodos que va tocando el circuito y eliminaremos cada nodo que ya hayamos visto antes en la lista. Así, si la lista dice u — v — w y decidimos eliminar a v, la lista de aristas quedará u — w, entendiéndose que en el camino hemos reemplazado las aristas (u,v) y (v,w) por la arista (u,w). Debido a la desigualdad triangular, estos cambios no incrementan el costo $c(C)$.

Una vez que hemos eliminado los nodos repetidos, el circuito resultante C' es hamiltoniano, y su costo está acotado por $c(C') \leq c(C) \leq 2 \cdot c(T) \leq 2 \cdot c^*$, con lo cual tenemos una 2-aproximación. Todos los algoritmos que hemos usado son de tiempo polinomial.

7.3. Recubrimiento de Conjuntos

Suponga que tiene conjuntos $S_1, \ldots, S_r$, con traslape para que sea interesante, y consideremos su unión, $S = \cup_{i=1}^{r} S_i$, llamando $n = |S|$. El problema de decisión NP-completo de *recubrimiento de conjuntos* es, dado además un k, determinar si hay k de esos conjuntos S_i cuya unión es S. El problema de optimización es encontrar el mínimo k. Esto modela problemas como comprar el mínimo número de discos que incluyen todas las canciones de un cierto artista.

Veremos que este problema admite una $\ln(n)$-aproximación. La técnica es un sencillo enfoque avaro: Elegimos primero el S_i que cubra más elementos de S (al comienzo, esto es simplemente el mayor conjunto S_i). Luego sacamos sus elementos de S, $S \leftarrow S - S_i$, e iteramos. Claramente esto da una solución de tiempo polinomial.

Para ver que esta solución es una $\ln(n)$-aproximación, consideremos que k es el tamaño de la solución óptima. Como el algoritmo óptimo cubre S con k conjuntos S_i, al menos uno de ellos debe ser de tamaño $\geq \frac{n}{k}$. Como nuestro algoritmo parte eligiendo el máximo conjunto, digamos S_x, este conjunto tiene tamaño al menos $\frac{n}{k}$. Eso significa que, luego de elegir el

primer conjunto, nuestro algoritmo deja S de tamaño a lo más $n(1-\frac{1}{k})$.

Lo que resta de S, $S-S_x$, también es cubierto con los k conjuntos elegidos originalmente por el óptimo, por lo tanto alguno de ellos debe cubrir al menos una fracción de $\frac{1}{k}$ de $S-S_x$. Nuevamente, nuestra aproximación elegirá entonces en el segundo paso un conjunto que cubra al menos esa fracción de $S-S_x$, por lo cual el nuevo tamaño del conjunto restante S será a lo más $n(1-\frac{1}{k})^2$.

Siguiendo este razonamiento, luego de m iteraciones, S será de tamaño a lo más $n(1-\frac{1}{k})^m$, y para que esta cota llegue a 1 (con lo cual el algoritmo termina en un paso más), basta que

$$m \;=\; -\frac{\log n}{\log(1-\frac{1}{k})} \;=\; \frac{\log n}{\log \frac{1}{1-\frac{1}{k}}} \;\leq\; k \ln n,$$

donde al final usamos que $\ln(1+\epsilon) \geq \frac{\epsilon}{1+\epsilon}$. Por lo tanto, el algoritmo aproximado no elige más de $1 + k \ln n$ conjuntos. Deteniéndonos en $m = k \ln \frac{n}{k}$, nos quedan k elementos por cubrir, los que necesitan a lo sumo k pasos más. Esto nos da una cota levemente mejor, $k(1 + \ln \frac{n}{k})$.

7.4. Llenar la Mochila

Finalmente, veremos un esquema de aproximación completamente polinomial. El *problema de la mochila* (también llamado *suma de subconjuntos*), parte de un multiconjunto de enteros positivos $X = \{x_1, \ldots, x_n\}$, y un tope t. En su versión de decisión, se pregunta si es posible encontrar un subconjunto de X cuya suma sea exactamente t. En la versión de optimización, queremos un conjunto $X' \subseteq X$ de suma máxima pero sin exceder t (la analogía es que se llena una mochila lo más posible sin exceder su capacidad).

Partamos con una solución exacta, aunque de tiempo exponencial. En el paso i, habremos generado una lista L_i que contiene, en forma creciente, todos los pesos que se pueden sumar con subconjuntos de $\{x_1, \ldots, x_i\}$, pero sin exceder t. Claramente el mayor elemento de L_n es la solución óptima. Inicialmente $L_0 = \{0\}$. Luego, dado L_{i-1}, podemos generar L_i mediante (a) generar $L'_{i-1} = \{v + x_i, v \in L_{i-1}\}$, es decir, agregar x_i a todas las soluciones de L_{i-1}; (b) unir $L_i \leftarrow L_{i-1} \cup L'_{i-1}$ eliminando repetidos; (c) eliminar los valores de L_i mayores a t (los tres pasos se pueden hacer simultáneamente en dos pasadas secuenciales simultáneas sobre L_{i-1}). La intuición es que, en el paso i, podemos o no usar x_i, por lo tanto consideramos agregarlo y no agregarlo a cada solución previa.

En esta solución exacta, el largo de L_i puede llegar a ser 2^i, por lo que el costo total del algoritmo es $O(2^n)$. Generaremos ahora una solución aproximada de tiempo polinomial.

La idea es que eliminaremos valores consecutivos de cada L_i que sean demasiado cercanos. Diremos que si $z < y$ son dos valores en L_i, entonces z *representa* y si

$$\frac{y}{1+\delta} \;\leq\; z \;<\; y,$$

donde δ es un parámetro que definiremos luego. Es decir, z es menor que y pero no está tan lejos. Cuando encontremos una situación así, eliminaremos y, dado que z está suficientemente

cerca. Note que no es buena idea eliminar z en vez de y, pues podría ser que y exceda t y nos quedemos sin ninguna de las dos.

Modificaremos el algoritmo exacto de manera que, luego de producir cada nueva lista L_i, le hagamos una pasada eliminando elementos que pueden ser representados por un elemento previo. Si $L_i = \{l_1, \ldots, l_r\}$, no podemos eliminar l_1, pero tomaremos $last \leftarrow l_1$ como un elemento que puede representar a otros. Así, eliminaremos l_2, l_3, ... hasta que encontremos un $l_k > (1+\delta) \cdot last$, el cual ya no puede ser representado con $last$. Entonces l_k será el siguiente elemento que sobrevive a la purga. Tomaremos $last \leftarrow l_k$ y continuaremos de la misma forma hasta procesar toda la lista L_i.

Con esta purga, el algoritmo resulta ser aproximado, ya que podemos perder el valor máximo de L_n, pero éste estará representado por algún otro valor algo menor. ¿Cuánto menor? Supongamos que el máximo z^* ya está presente en L_1, pero es eliminado porque es representado por un valor menor, $z_1 \geq \frac{z^*}{1+\delta}$. En L_2, z_1 es a su vez eliminado porque es representado por otro valor menor, $z_2 \geq \frac{z_1}{1+\delta}$, y así. Al final, tendremos en L_n un valor $z_n \geq \frac{z^*}{(1+\delta)^n}$. Como lo que deseamos es una $(1+\epsilon)$-aproximación para un ϵ dado, queremos garantizar que $z_n \geq \frac{z^*}{1+\epsilon}$, por lo que debe cumplirse que $(1+\delta)^n \leq 1+\epsilon$. Usamos esta desigualdad para definir δ precisamente como $\delta = (1+\epsilon)^{1/n} - 1$.

La pregunta final es si, usando este valor de δ, el algoritmo resultante es de tiempo polinomial. Note que, como el primer valor de cualquier L_i es a lo menos 1, cada valor siguiente en L_i luego de la purga es a lo menos $(1+\delta)$ veces mayor que el anterior, y el último valor es a lo sumo t, tenemos que L_i tiene a lo sumo $1 + \log_{1+\delta} t$ elementos, es decir,

$$|L_i| \quad \leq \quad 1 + \frac{\log t}{\log(1+\delta)} \quad = \quad 1 + \frac{n \log t}{\log(1+\epsilon)}.$$

Como $\frac{\epsilon}{1+\epsilon} \leq \ln(1+\epsilon) \leq \epsilon$, esto es $|L_i| \leq 1 + \frac{(1+\epsilon) n \ln t}{\epsilon} = O(\frac{n \log t}{\epsilon})$. El costo total de las n iteraciones es entonces $O(\frac{n^2 \log t}{\epsilon})$, lo cual es polinomial tanto en la entrada como en $\frac{1}{\epsilon}$.

7.5. Búsqueda Exhaustiva

Antes de terminar, vale la pena hacer notar que un método inteligente de búsqueda exhaustiva puede encontrar la solución óptima en un tiempo que, si bien es exponencial en el peor caso, puede resultar mucho menor, incluso práctico, en muchos casos de la vida real.

El método llamado *backtracking* consiste en construir todas las posibles soluciones, probando en cada paso de la construcción todas las opciones posibles. Así se genera un árbol virtual donde cada nodo interno es una solución parcial y cada hoja es una solución completa, y nos quedamos con la hoja óptima. Podemos evitar generar el subárbol de un nodo cuando detectamos que es imposible obtener una solución válida a partir de las decisiones ya tomadas. Una sofisticación llamada *branch and bound* permite también cortar la generación del subárbol de un nodo interno cuando puede predecir que ninguna hoja de ese subárbol será competitiva contra la mejor solución generada hasta el momento.

Tomemos nuevamente el problema de la mochila. Podemos almacenar una variable global con la mejor solución z^* conocida hasta el momento, y explorar, para cada variable x_i, si la incluimos o no en la solución. Partimos entonces con la solución $z^* = 0$, correspondiente a no incluir ningún x_i, e invocamos una función $probar(1, t)$. La función $probar(i, p)$ genera la mejor solución sumando números de $\{x_i, \ldots, x_n\}$ sin exceder p. Por lo tanto, prueba no incluir x_i, invocando $probar(i + 1, p)$, e incluir x_i, invocando $probar(i + 1, p - x_i)$. Cada vez que llegamos a $probar(n + 1, p)$, si $p \geq 0$, recalculamos $z^* \leftarrow \text{máx}(z^*, t - p)$.

Esta recursión realmente genera exhaustivamente todas las 2^n soluciones, válidas e inválidas. El backtracking puede mejorarse notando que, si invocamos $probar(i, p)$ con $p < 0$, para cualquier i, ya no tenemos chance de obtener una solución válida porque hemos excedido la capacidad t de la mochila. Por lo tanto, podemos abortar la recursión en esos casos, evitando generar grandes subárboles inútiles. La solución resultante de esta poda es similar a la solución exacta de las listas L_i que vimos como preludio a la aproximación. Sin embargo, podemos mejorarla más con un enfoque de branch and bound. Por ejemplo, podemos precalcular todas las sumas $S_i = x_i + \ldots + x_n$, y abortar la búsqueda en $probar(i, p)$ si $t - p + S_i \leq z^*$, es decir, ya no tenemos chance de alcanzar el mejor z^* conocido incluso agregando todos los x_i que tenemos por delante.

Asimismo, la programación dinámica es una forma de acelerar la búsqueda exhaustiva, cuando puede aplicarse. Para el mismo ejemplo de la mochila, si t no es demasiado grande, una solución es calcular la matriz $M[i, p]$, que da la mejor solución con $\{x_1, \ldots, x_i\}$ y tope p. Por lo tanto, calculamos cada celda $M[i, p] \leftarrow \text{máx}(M[i - 1, p], x_i + M[i - 1, p - x_i])$ (el segundo término sólo si $x_i \leq p$), $M[0, p] \leftarrow 0$ y $M[i, 0] \leftarrow 0$, en tiempo constante. Así encontramos el óptimo $M[n, t]$ en tiempo $O(nt)$, lo que es exponencial en el tamaño de la entrada (que es $O(n \log t)$) pero, como se dijo, puede ser aceptable si t no es muy grande.

7.6. Ficha Resumen

- Conceptos de algoritmos aproximados.
- Recubrimiento de vértices: 2-aproximable, sin y con pesos.
- Viajante de comercio: 2-aproximable si los costos son métricos, inaproximable si no.
- Suma de subconjuntos: $\log(n)$-aproximable.
- Llenar la mochila: $(1 + \epsilon)$-aproximable completamente polinomial.
- Búsqueda exhaustiva usando branch and bound.

7.7. Material Suplementario

El material de este capítulo está basado casi completamente en Cormen et al. [CLRS01, cap. 35], excepto por la búsqueda exhaustiva y el análisis del recubrimiento de conjuntos. Muchos otros libros contienen material de algoritmos aproximados también. Mehlhorn y Sanders [MS08, sec. 12.1 y 12.2] ven los problemas de la mochila y de scheduling como ejemplos de aproximaciones basadas en programación lineal y de algoritmos avaros. Baase [Baa88, sec. 9.3 a 9.6] presenta varios algoritmos aproximados, incluyendo los problemas de la mochila, bin packing (una variante de la mochila), y coloreo de grafos. Manber [Man89, sec. 11.5.2] ve recubrimiento de vértices, bin packing, y el viajante de comercio, incluyendo la 1,5-aproximación (el llamado algoritmo de Christofides). Levitin [Lev07, sec. 12.3] discute con detalle el viajante de comercio (incluyendo Christofides) y la mochila. Kleinberg y Tardos [KT06, cap. 11] estudian en detalle los problemas de scheduling, ubicación de servidores, recubrimiento de conjuntos, recubrimiento de vértices (con otra solución), caminos disjuntos y mochila (donde los elementos tienen pesos y valores). Se enfocan en detalle en programación lineal y el método llamado de "pricing" como técnicas generales. Dasgupta et al. [DPV08, sec. 5.4 y 9.2] explican sucintamente recubrimiento de conjuntos, de vértices, mochila (con otra solución basada en reescalar los valores y usar programación dinámica), y un interesante problema de clustering. Finalmente, Lee et al. [LTCT05, cap. 9] estudian con bastante detalle recubrimiento de vértices, viajante de comercio (y una variante), ubicación de servidores, bin packing, conjunto independiente maximal, mochila, ruteo en grafos, y problemas relevantes en bioinformática como alineamiento múltiple y ordenar mediante transposiciones.

Brassard y Bratley [BB88, sec. 6.6], Aho et al. [AHU83, sec. 10.4], Weiss [Wei95, sec. 10.5], Mehlhorn y Sanders [MS08, sec. 12.4] (en menor medida), Sedgewick [Sed92, cap. 44] (brevemente), Manber [Man89, sec. 11.5.1], y Levitin [Lev07, sec. 12.1 y 12.2] (con bastante detalle) describen el backtracking y el branch and bound, con variados ejemplos. Los tres primeros describen también la *poda alfa-beta*, que es útil especialmente en juegos.

Aho et al. [AHU83, sec. 10.5], Mehlhorn y Sanders [MS08, sec. 12.5 y 12.6], Kleinberg y Tardos [KT06, cap. 12] y Dasgupta et al. [DPV08, sec. 9.3] describen otras heurísticas para encontrar valores cercanos al óptimo en casos muy complejos, cuando ya no se tiene esperanza de garantizar una aproximación: búsqueda local, algoritmos evolutivos, y otros.

Finalmente, Lee et al. [LTCT05, sec. 9.12] describen el interesante concepto de NPO-completitud, que define la jerarquía de complejidad para problemas de optimización y permite demostrar que ciertos problemas de optimización son imposibles de aproximar (siempre que $P \neq NP$). Entre estos problemas se encuentran la versión de optimización de satisfactibilidad, la programación entera, y encontrar el circuito hamiltoniano más caro y más barato (este último es, precisamente, el problema del viajante de comercio).

Otras fuentes online de interés:

- `www.cc.gatech.edu/fac/Vijay.Vazirani/book.pdf`

- www.designofapproxalgs.com/book.pdf
- pdfs.semanticscholar.org/4439/63a150ddce5bde1f0e5930971f66d5bffe51.pdf
- www.cs.cmu.edu/~avrim/451f12/lectures/lect1106.pdf
- pages.cs.wisc.edu/~shuchi/courses/880-S07/scribe-notes/all-in-one.pdf
- www.cs.princeton.edu/~wayne/cs423/lectures/approx-alg-4up.pdf
- theory.stanford.edu/~tim/w16/l/l15.pdf .. l17.pdf
- www.youtube.com/watch?v=MEz1J9wY2iM
- www.youtube.com/watch?v=4q-jmGrmxKs
- www.youtube.com/watch?v=zM5MW5NKZJg

Capítulo 8

Algoritmos Paralelos

Hasta ahora hemos considerado que los algoritmos son *secuenciales*, es decir, ejecutan una instrucción tras otra. Desde que la velocidad de los procesadores ha dejado de duplicarse cada dos años por límites físicos, la Ley de Moore se ha traducido en el incremento del número de procesadores que pueden trabajar paralelamente. Asimismo, hay cada vez un mayor interés en usar GPUs (Graphic Processing Units, originalmente diseñadas para procesar instrucciones gráficas) para implementar soluciones a diversos problemas usando paralelismo masivo tipo SIMD (Single Instruction, Multiple Data) en procesadores más o menos convencionales.

Si bien los compiladores pueden paralelizar automáticamente algunos programas, no siempre lo logran al máximo. El diseño de algoritmos paralelos requiere, en muchos casos, eliminar dependencias secuenciales aparentes para poder partir un problema en subproblemas que se puedan resolver al mismo tiempo.

Si bien existen muchos modelos de computación paralela (multithreaded, distribuida, sincrónica, etc.) los fundamentos algorítmicos son similares, por lo cual vamos a trabajar en base a un modelo llamado PRAM (Parallel RAM), que tiene la ventaja de ser sencillo y dejarnos concentrar en lo algorítmico. Este modelo es cercano a la programación de GPUs.

8.1. El Modelo PRAM

En este modelo tenemos un número arbitrario de procesadores. Todos tienen el mismo programa, y todos ejecutan la misma instrucción al mismo tiempo, en sincronía perfecta. Trabajan sobre una memoria global compartida. El input se encuentra en esta memoria y el output se deja ahí también.

Una variable especial, *pid*, entrega un valor distinto según qué procesador la lea: al procesador i le devuelve i. Esto permite que los procesadores hagan cosas distintas, por ejemplo, para poner en cero un arreglo $A[1..n]$ teniendo n procesadores (numerados de 1 a n), basta con que todos ejecuten la instrucción $A[pid] \leftarrow 0$. El *pid* también permite que los procesadores tengan variables locales x mediante guardarlas en un arreglo global $X[1..n]$, e interpretar

que $x = X[pid]$ (normalmente hablaremos de variables locales a cada procesador, entendiendo esta forma de implementarlas). Finalmente, el *pid* permite que algunos procesadores se abstengan de realizar una acción. Por ejemplo, para poner en 1 las posiciones pares de nuestro arreglo A, podemos decir **if** *pid* mód $2 = 0$ **then** $A[pid] \leftarrow 1$. Lo que ocurre en este caso es que los procesadores que dan falso en la condición *se inhabilitan* hasta que los demás terminen de ejecutar el **if**. No pueden ignorar el **if** y proseguir con la siguiente instrucción. Esto debe ser así para que todos trabajen sincronizadamente. En una instrucción **if-then-else**, algunos se inhabilitan mientras se ejecuta la parte del **if** y los otros se inhabilitan luego, cuando los primeros ejecutan la parte del **else**.

Esto puede llevarse a la recursión también, en la que debe mantenerse la sincronización. Supongamos que en $mergesort(i, j)$, ejecutada por los procesadores i a j, queremos que se partan en dos mitades y se invoquen recursivamente en los subarreglos. Podemos decir: $m \leftarrow \lfloor \frac{i+j}{2} \rfloor$ y luego: **if** $pid \leq m$ **then** $j \leftarrow m$ **else** $i \leftarrow m + 1$. Note que m, i y j son variables locales. Finalmente, todos los procesadores se invocan recursivamente con $mergesort(i, j)$, lo que hará que la mitad de ellos se aboque al subarreglo izquierdo, $[i..m]$, y la otra mitad al derecho, $[m + 1..j]$, si bien continuarán ejecutando exactamente las mismas instrucciones.

El modelo PRAM tiene tres submodelos, según cómo se permite a los procesadores acceder a la memoria compartida. Los listamos de más a menos restrictivos (un algoritmo en un modelo más restrictivo es más conveniente, pues requiere menos poder).

EREW (Exclusive Read, Exclusive Write) Dos procesadores no pueden leer ni escribir una misma celda al mismo tiempo.

CREW (Concurrent Read, Exclusive Write) Dos procesadores pueden leer una misma celda al mismo tiempo, pero no escribirla.

CRCW (Concurrent Read, Concurrent Write) Dos procesadores pueden leer y escribir una misma celda al mismo tiempo. Hay submodelos de CRCW según lo que ocurra con la celda escrita. El más restrictivo es el CRCW *común*, en que sólo pueden escribir al mismo tiempo si escriben el mismo valor. Una segunda posibilidad es el CRCW *arbitrario*, en que cuando varios escriben una misma celda simultáneamente, lo que queda escrito es alguno de los valores, elegido arbitrariamente. Existen otros modelos incluso más convenientes, pero no se toman muy en serio en la práctica.

Un ejemplo donde se hace clara la diferencia entre el poder de EREW y de CREW es cuando queremos copiar a todas las celdas de un arreglo $A[1..n]$ el valor de $A[1]$. En el modelo CREW, podemos simplemente hacer $A[pid] \leftarrow A[1]$, en tiempo constante. En el modelo EREW, en cambio, debemos hacer $\log n$ iteraciones donde vamos duplicando el valor: en el paso $\ell = 0, \ldots, (\log n) - 1$, hacemos **if** $pid \leq 2^\ell$ **then** $A[pid + 2^\ell] \leftarrow A[pid]$.

Un ejemplo del poder del modelo CRCW (común) es el algoritmo que encuentra el máximo en un arreglo $A[1..n]$ usando $\frac{n(n-1)}{2}$ procesadores en tiempo constante. Llamemos a los procesadores (i, j), con $i < j$. Primero, n de estos procesadores inicializan un arreglo $R[1..n]$

en 1. Entonces, el procesador (i, j) compara $A[i]$ con $A[j]$. Si $A[i] \leq A[j]$, escribe $R[i] \leftarrow 0$, si no, escribe $R[j] \leftarrow 0$. Luego de esto, sólo el máximo $A[k]$ conserva su celda $R[k] = 1$ (frente a un empate, gana el mayor k). Entonces n procesadores miran las celdas de R y el que encuentra el 1 entrega el índice k como respuesta.

8.2. Modelo de Costos

En los algoritmos paralelos importa no sólo el tiempo de ejecución, sino cuántos procesadores necesitamos para lograrlo, es decir, cuán eficientemente usamos los procesadores.

Tendremos n en general como el tamaño del input, y usaremos p para denotar la cantidad de procesadores. Entonces $T(n, p)$ será el tiempo que toma el algoritmo con un input de tamaño n y usando p procesadores. El sentido de $T(n, 1)$ será especial: representará el tiempo *del mejor* algoritmo secuencial, no el de mi algoritmo usando 1 procesador. Esto importa para medir cuán bien escala algoritmo paralelo con p. La medida

$$S(n, p) \quad = \quad \frac{T(n, 1)}{T(n, p)}$$

se llama *speedup*, e indica cuánto se acelera el algoritmo al usar p procesadores (en general omitiremos la notación $O(\cdot)$ en este capítulo). Como el numerador es el mejor algoritmo secuencial, evitamos engañarnos con algoritmos paralelos que parezcan mejorar mucho al usar más procesadores por la mera razón de estar paralelizando un mal algoritmo. Por ejemplo, es fácil paralelizar bien un algoritmo de sorting cuadrático: cada procesador se compara con todos, obtiene el número t de valores menores a su celda, y luego escribe su valor en la posición $t + 1$. Con n procesadores, ordenamos en tiempo n, mientras que con 1 procesador nos tomaría n^2. El speedup nos daría n, que parecería muy bueno. Sin embargo, la idea no es tan buena cuando sabemos que secuencialmente se puede ordenar en tiempo $n \log n$: usamos n procesadores y el tiempo no baja n veces, sino sólo $\log n$ veces. El verdadero speedup es sólo $S(n, n) = \log n$.

Está claro entonces que $S(n, p) \leq p$, pues si con p procesadores obtuviéramos $T(n, 1) > pT(n, p)$, entonces podríamos simular los p procesadores usando uno solo y obtendríamos tiempo $pT(n, p)$, superando al mejor algoritmo secuencial.

Una medida relacionada con el speedup es la *eficiencia*,

$$E(n, p) \quad = \quad \frac{S(n, p)}{p} \quad = \quad \frac{T(n, 1)}{pT(n, p)},$$

donde vale que $0 \leq E(n, p) \leq 1$. La eficiencia indica qué fracción de la capacidad de los procesadores se está usando (con respecto al mejor algoritmo secuencial). Lo ideal es usar una fracción constante de la capacidad (que, como ignoraremos la notación $O(\cdot)$, veremos como eficiencia 1). Por ejemplo, la eficiencia de nuestro algoritmo de ordenamiento es sólo $E(n, n) = \frac{\log n}{n}$, sumamente baja: estamos usando demasiados procesadores para la reducción en tiempo

que estamos obteniendo. Lo mismo ocurre con nuestro algoritmo CRCW para encontrar el máximo: a pesar de que su tiempo es sólo $T(n, n^2) = 1$, su eficiencia es $E(n, n^2) = \frac{n}{n^2 \cdot 1} = \frac{1}{n}$.

En la mayoría de los algoritmos es factible que, si lo diseñé para p procesadores pero tengo solamente $\frac{p}{k}$ disponibles, cada procesador haga el trabajo de k procesadores "virtuales". Así, se obtiene $T(n, \frac{p}{k}) = k\,T(n, p)$. Es entonces conveniente diseñar los algoritmos usando todos los procesadores que se desee, entendiendo que no significa que necesitemos esa cantidad, sino que en caso de tener menos el tiempo escalará en forma correspondiente. Así, cuando está claro que el algoritmo escala de esta manera, se suele presentar su tiempo usando dos medidas:

- $T(n)$ ("time") es el tiempo que demora el algoritmo usando todos los procesadores que desee. Indica el grado de paralelización que se puede alcanzar: por más que tenga más procesadores, el tiempo no puede bajar de $T(n)$.
- $W(n)$ ("work") es el trabajo total que realiza el algoritmo, sumando todos los procesadores pero sin contar el tiempo que pasan inhabilitados. Indica el tiempo que demoraría ejecutar el algoritmo con un solo procesador que simulara a todos. Un algoritmo sólo puede tener eficiencia máxima si $W(n) = T(n, 1)$.

Los algoritmos que se describen así pueden emularse con cualquier cantidad deseada p de procesadores, en tiempo

$$T(n, p) \quad = \quad T(n) + \frac{W(n)}{p}.$$

Note que esto significa que no vale la pena usar más de

$$p^* \quad = \quad \frac{W(n)}{T(n)}$$

procesadores. Con esa cantidad ideal, el tiempo es el mejor posible:

$$T(n, p^*) \quad = \quad T(n)$$

y la eficiencia también:

$$E(n, p^*) \quad = \quad \frac{T(n, 1)}{p^* T(n, p^*)} \quad = \quad \frac{T(n, 1)}{W(n)}$$

(que es 1 si el algoritmo básico es el mejor posible). Si usamos menos de p^* procesadores, la eficiencia sigue siendo alta pero el tiempo $T(n, p)$ aumenta, a $\frac{W(n)}{p}$. Si usamos más de p^* procesadores, el tiempo sigue siendo $T(n)$, pero la eficiencia empieza a decrecer, a $\frac{T(n,1)}{p\,T(n)}$.

El *Lema de Brent* establece que un algoritmo EREW donde se pueda calcular en tiempo constante qué procesadores están activos en cada momento, puede siempre expresarse como $T(n)$ y $W(n)$ y obtener el $T(n, p)$ que vimos para cualquier p. El lema es algo técnico, por lo cual seguiremos describiendo cómo los algoritmos que vemos permiten hacer esto.

8.3. Sumando un Arreglo

Comenzaremos con un problema muy sencillo que nos permite ilustrar los puntos anteriores. Consideremos un arreglo $A[1..n]$, del que tenemos que obtener la suma de todos sus elementos, $S = \sum_{i=1}^{n} A[i]$ (podemos usar cualquier otra operación asociativa). Consideremos que tenemos n procesadores. Lo que podemos usar, entonces, es una estructura de tipo torneo de tenis en $\lceil \log n \rceil$ pasos, numerados 0 a $\lceil \log n \rceil - 1$. En el paso ℓ, los procesadores ejecutan

$$\textbf{if } (pid - 1) \text{ mód } 2^{\ell+1} = 0 \ \wedge \ pid + 2^{\ell} \leq n \textbf{ then } A[pid] \leftarrow A[pid] + A[pid + 2^{\ell}].$$

Entonces, en el paso $\ell = 0$, habremos hecho $A[1] \leftarrow A[1] + A[2]$, $A[3] \leftarrow A[3] + A[4]$, $A[5] \leftarrow A[5] + A[6]$, $A[7] \leftarrow A[7] + A[8]$, etc. En el siguiente paso, $\ell = 1$, haremos $A[1] \leftarrow A[1] + A[3]$, $A[5] \leftarrow A[5] + A[7]$, etc. En el paso $\ell = 2$, haremos $A[1] \leftarrow A[1] + A[5]$, etc. Es fácil ver que, cuando terminemos, la suma estará en $A[1]$.

Éste es un algoritmo EREW. Lo hemos descrito para n procesadores, donde toma tiempo $T(n, n) = \log n$ y su eficiencia es $E(n, n) = \frac{n}{n \log n} = \frac{1}{\log n}$.

¿Puede mejorarse la eficiencia usando menos procesadores? Si tenemos p procesadores, cada uno puede primero sumar $\frac{n}{p}$ números, y luego los p procesadores suman los p números usando el algoritmo paralelo visto. El tiempo total es $T(n, p) = \frac{n}{p} + \log p$. En términos de orden, esto es equivalente a $T(n, p) = \frac{n}{p} + \log n$. Por ello, podemos describir el costo del algoritmo en forma genérica en términos de tiempo y trabajo: $T(n) = \log n$ y $W(n) = n$ (pues en total se realizan $\frac{n}{2} + \frac{n}{4} + \ldots + 1 < 2n$ sumas). El número ideal de procesadores para este algoritmo es entonces $p^* = \frac{n}{\log n}$, en el cual el tiempo aún será $T(n, p^*) = \log n$ y la eficiencia habrá mejorado a $E(n, p^*) = 1$.

8.4. Parallel Prefix

Consideremos un operador asociativo $+$. Dado un arreglo $A[1..n]$, el problema de *parallel prefix* es el de reescribir $A[i] \leftarrow \sum_{j=1}^{i} A[j]$. Esto puede resolverse fácilmente en $T(n, 1) = n$ mediante hacer, para $i = 2, \ldots, n$, $A[i] \leftarrow A[i - 1] + A[i]$. Pero este algoritmo es intrínsicamente secuencial, ¿puede paralelizarse?

8.4.1. Un método recursivo

Veamos primero una técnica recursiva, que se invocará como $pp(1, n)$. En general, $pp(a, b)$ calculará el parallel prefix correcto para $A[a..b]$, es decir, $A[k] \leftarrow \sum_{j=a}^{k} A[j]$ para todo $a \leq k \leq b$. Para resolver $pp(a, b)$ con los procesadores numerados a a b, calcularemos $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$. Los procesadores a a m calcularán $pp(a, m)$, mientras que los procesadores $m+1$ a b calcularán $pp(m+1, b)$. Una vez terminadas ambas mitades (que se ejecutan simultáneamente), notamos que los valores calculados en $A[a..m]$ ya están correctos, mientras que a los que están en

$A[m+1..b]$ sólo necesitan que se les haga $A[k] \leftarrow A[m] + A[k]$ (todos a la vez, en una única instrucción **if** $pid > m$ **then** $A[pid] \leftarrow A[m] + A[pid]$).

Éste es un algoritmo CREW, pues al corregir la parte derecha del arreglo todos leen la misma celda $A[m]$. ¿Podemos convertirlo en EREW? Sí, y en este caso sin costo (asintótico) adicional. Calculemos, a la par de A, un arreglo B donde $pp(a, b)$ dejará escrito $B[k] = A[b]$ para todo $a \leq k \leq b$. Entonces, al volver de las invocaciones $pp(a, m)$ y $pp(m+1, b)$ haremos

$$\textbf{if } pid > m \textbf{ then } A[pid] \leftarrow B[pid - (m - a + 1)] + A[pid],$$

lo cual es EREW. También, para reestablecer el invariante sobre B, haremos

$$\begin{aligned}
&\textbf{if } pid > m \textbf{ then } B[pid] \leftarrow B[pid - (m - a + 1)] + B[pid] \\
&\textbf{if } pid < m \textbf{ then } B[pid] \leftarrow B[pid + (m - a + 1)] \\
&\textbf{if } pid = m \textbf{ then } B[pid] \leftarrow B[b].
\end{aligned}$$

Ahora bien, este algoritmo tiene tiempo $T(n, n) = \log n$ y eficiencia $E(n, n) = \frac{1}{\log n}$. Hemos logrado paralelizar la solución, pero ¿podemos mejorar la eficiencia, como lo hicimos para sumar n elementos? Esta vez no podemos porque, si bien $T(n) = \log n$, el trabajo que se hace es $W(n) = n \log n$. Esto se deduce de que, para una invocación de pp de tamaño n, el trabajo es $t(n) = 2\, t(\frac{n}{2}) + n$, este último n para corregir los valores de la mitad derecha. Por lo tanto, el número óptimo de procesadores es $p^* = n$, con eficiencia $E(n, p^*) = \frac{1}{\log n}$. Dicho de otro modo, no podemos tener eficiencia 1 porque $W(n)$ es mayor que $T(n, 1)$.

8.4.2. Un método más eficiente

Podemos, en cambio, diseñar una paralelización completamente distinta, que sí nos entregará una mejor eficiencia. La idea es que primero realizamos un torneo de tenis "ascendente", donde en el paso ℓ cada celda que sea múltiplo de $2^{\ell+1}$ suma a su contenido la que está a distancia 2^ℓ hacia atrás. Ahora un torneo de tenis "descendente" completa el cálculo: cada celda múltiplo de $2^{\ell+1}$ le agrega su contenido a la que está a distancia 2^ℓ hacia adelante. Concretamente, para $\ell = 0, \ldots, \lceil \log n \rceil - 1$, haremos

$$\textbf{if } pid \mod 2^{\ell+1} = 0 \ \wedge \ pid > 2^\ell \textbf{ then } A[pid] \leftarrow A[pid - 2^\ell] + A[pid]$$

y luego, para $\ell = \lceil \log n \rceil - 1, \ldots, 0$, haremos

$$\textbf{if } pid \mod 2^{\ell+1} = 0 \ \wedge \ pid \leq n - 2^\ell \textbf{ then } A[pid + 2^\ell] \leftarrow A[pid] + A[pid + 2^\ell].$$

El algoritmo funciona porque, al terminar la fase ascendente, las celdas que son múltiplo de 2^ℓ (y no de $2^{\ell+1}$) quedan sumadas con las $2^\ell - 1$ celdas anteriores. Luego, en la fase descendente, se les suma la celda a distancia 2^ℓ hacia atrás, la cual es múltiplo de $2^{\ell+1}$ y, por hipótesis inductiva, ya tiene su valor correcto calculado.

Consideremos la iteración ascendente y $n = 8$. En el paso $\ell = 0$, haremos $A[2] \leftarrow A[1] + A[2]$, $A[4] \leftarrow A[3] + A[4]$, $A[6] \leftarrow A[5] + A[6]$, y $A[8] \leftarrow A[7] + A[8]$. En el paso $\ell = 1$ haremos $A[4] \leftarrow A[2] + A[4]$ y $A[8] \leftarrow A[6] + A[8]$. En el paso $\ell = 2$ haremos $A[8] \leftarrow A[4] + A[8]$. Ahora la fase descendente comienza con $\ell = 2$, en que no hace nada. Con $\ell = 1$ hace $A[6] \leftarrow A[4] + A[6]$. Con $\ell = 0$ hace $A[3] \leftarrow A[2] + A[3]$, $A[5] \leftarrow A[4] + A[5]$, y $A[7] \leftarrow A[6] + A[7]$. Puede verificarse que las sumas se han realizado correctamente.

Este algoritmo es EREW, con $T(n) = \log n$ y $W(n) = n$. Se puede correr con $p^* = \frac{n}{\log n}$ procesadores y obtener tiempo $T(n, p^*) = \log n$ y eficiencia $E(n, p^*) = 1$, mediante hacer que cada procesador se encargue de $\frac{n}{p^*}$ celdas consecutivas de A.

8.5. List Ranking

Suponga que tiene una lista implementada en un arreglo $A[1..n]$, es decir, cada elemento $A[i]$ tiene indicado en $N[i]$ cuál es su siguiente elemento, con el último elemento de la lista indicado como $N[i] = 0$. Nos gustaría poner el arreglo A en orden lineal, pero para ello necesitamos saber cuál es su posición. Decimos que el *rank* de una posición i es su distancia al final de la lista. Si conseguimos calcular todos los ranks en un arreglo R, entonces podremos poner A en orden lineal mediante simplemente $A[n + 1 - R[pid]] \leftarrow A[pid]$.

Pero ¿cómo calcular el rank de todos los elementos? Es fácil hacerlo secuencialmente, entrando en la lista recursivamente hasta hallar el último elemento y luego ir asignando los ranks en forma creciente a la vuelta de la recursión. Pero ¿puede paralelizarse?

Procederemos en $\log n$ iteraciones. En el paso ℓ, todos los elementos a distancia $\leq 2^{\ell+1}$ del final de la lista descubrirán su rank, y en vez de apuntar al siguiente de la lista, quedarán apuntando en N al elemento $2^{\ell+1}$ posiciones hacia adelante. Para ello, inicializaremos R con **if** $N[pid] = 0$ **then** $R[pid] \leftarrow 1$ **else** $R[pid] \leftarrow 0$. Luego haremos, para $\ell = 0, \ldots, \lceil \log n \rceil - 1$,

if $N[pid] \neq 0 \ \wedge \ R[N[pid]] \neq 0$ **then** $R[pid] \leftarrow R[N[pid]] + 2^\ell$
if $N[pid] \neq 0$ **then** $N[pid] \leftarrow N[N[pid]]$.

Consideremos la lista $8 \to 7 \to 6 \to 5 \to 4 \to 3 \to 2 \to \mathbf{1} \to 0$ (identificando los elementos con sus ranks, poniendo en bold los elementos que ya conocen sus ranks, y tomando el 0 como nulo). Al inicializarse, sólo el 1 conoce su rank. Luego de ejecutar para $\ell = 0$, pasamos a $8 \to 7 \to 6 \to 5 \to 4 \to 3 \to \mathbf{2} \to \mathbf{1} \to 0$. Sin embargo, al hacer que N apunte dos posiciones hacia adelante, nos quedan en la práctica dos listas, $8 \to 6 \to 4 \to \mathbf{2} \to 0$ y $7 \to 5 \to 3 \to \mathbf{1} \to 0$. Al ejecutar para $\ell = 1$ aprendemos nuevos ranks: $8 \to 6 \to \mathbf{4} \to \mathbf{2} \to 0$ y $7 \to 5 \to \mathbf{3} \to \mathbf{1} \to 0$. Al doblar N nuevamente, nos quedan cuadro listas, $8 \to \mathbf{4} \to 0$, $6 \to \mathbf{2} \to 0$, $7 \to \mathbf{3} \to 0$, y $5 \to \mathbf{1} \to 0$. En la última iteración, para $\ell = 2$, se aprenden todos los ranks que faltan.

El algoritmo resultante es EREW, con $T(n) = \log n$ y $W(n) = n \log n$, por lo que no puede lograrse eficiencia 1 mediante reducir la cantidad de procesadores.

8.6. Tour Euleriano

Así como el *list ranking* es una forma de escribir una lista en forma lineal en un arreglo, el *tour euleriano* es una forma de hacerlo para árboles (consideramos árboles en el sentido de grafos, es decir grafos no dirigidos, conexos y acíclicos). Un tour euleriano es un recorrido en profunidad del árbol que pasa por todos sus nodos, listando cada arista dos veces (una de ida y una de vuelta del recorrido). Es muy sencillo hacerlo con un recorrido DFS a partir de cualquier nodo, en tiempo $T(n, 1) = n$, pero ¿es posible paralelizarlo?

Consideremos la siguiente representación del árbol. Cada nodo i tiene una lista enlazada de las aristas que inciden en él. El puntero a la lista es $E(i)$, y cada elemento de la lista corresponde a una arista (i, j) para algún nodo j. La lista termina con el puntero 0. Para cada arista (i, j), $next(i, j)$ es el siguiente elemento de la lista de i. Asimismo, como las aristas no son dirigidas, la (i, j) también está representada como (j, i) en la lista del nodo j. Supondremos que existen punteros directos entre las dos orientaciones de cada arista.

Usaremos las dos notaciones de la misma arista, (i, j) y (j, i), para denotar en qué dirección las recorre el tour euleriano (de i hacia j en el primer caso, de j hacia i en el segundo). Así, podemos pensarlas como dos aristas dirigidas distintas. Es posible determinar, en paralelo, cuál será la arista que sigue a cualquier arista dirigida (i, j) en el tour euleriano. A esta siguiente arista la llamaremos $nextTour(i, j)$ y se cumple que

$$nextTour(i, j) \;=\; \begin{cases} next(j, i), & \text{si } next(i, j) \neq 0 \\ E(j), & \text{si no.} \end{cases}$$

Nótese que la expresión efectivamente trata a la lista de aristas que inciden en j como si fuera circular (si $next(j, i)$ es la última, volvemos al comienzo con $E(j)$). Así, $nextTour(i, j)$ nos da la arista de j que sigue a (j, i) en su lista, y recorre todos los otros vecinos de j antes de volver a entregar (j, i). Si j es una hoja, el resultado es (j, i) inmediatamente. Así, $nextTour(i, j)$ nos da un tour euleriano de todo el subárbol que parte en el nodo j y termina con la arista (j, i), que es la misma (i, j) con la que entramos pero en dirección opuesta.

Si tenemos, entonces, a todas las aristas (i, j) en un arreglo, podemos ejecutar un algoritmo de *list ranking* para determinar en qué posición del tour ubicar a cada arista (i, j). Para ello, primero inicializamos $N[k] \leftarrow nextTour(i, j)$, donde k representa la arista (i, j). El tour es circular: podemos partir de cualquier arista (i, j), poniendo su celda $N[k] \leftarrow 0$, y tendremos el tour que parte en (j, i) y termina luego de volver por (i, j). El costo de construir el tour euleriano es entonces igual al de hacer list ranking.

8.7. Ordenamiento

Veamos ahora otra primitiva importante: ordenar un arreglo $A[1..n]$. Veremos cómo adaptar MergeSort. La idea será particionar el arreglo por la mitad y ordenar recursivamente las dos mitades en paralelo. La parte compleja es entonces cómo hacer la mezcla de las dos mitades ya ordenadas, $A_1 = A[1..\frac{n}{2}]$ y $A_2 = A[\frac{n}{2} + 1..n]$.

El mecanismo secuencial de mezclado no parece fácil de paralelizar, pero hay en realidad varias formas de hacerlo. Veremos dos.

8.7.1. Un algoritmo EREW

Una particularmente elegante es la siguiente. La mitad de los n procesadores se dedicará a mezclar las posiciones *pares* de A_1 y A_2, dejando el resultado en P. La otra mitad mezclará las posiciones *impares* y dejará el resultado en I. Este mezclado se hará recursivamente. Una vez que tengamos P e I, la mezcla de ambas es sumamente simple: $I[1]$ es el menor de los dos mínimos $A_1[1]$ y $A_2[1]$, por lo que se ubica en $A[1]$. Luego, para $i \geq 1$, los elementos $P[i]$ e $I[i+1]$ deben ser escritos en $A[2i..2i+1]$, en el orden que les corresponda (es decir, los comparamos y los escribimos).

La razón de esto es la siguiente: $x = P[i]$ es el i-ésimo elemento de la mezcla de los elementos pares de las secuencias originales. Por lo tanto, x es mayor que otros $i-1$ de aquellos elementos pares. Pero cada uno de aquellos elementos pares, incluido x, estaba precedido de un elemento impar menor, por lo cual x también es mayor que i elementos impares. Por lo tanto, la posición de x en la secuencia completa mezclada es $> (i-1)+(i) = 2i-1$. Con un razonamiento similar, $y = I[i+1]$ es mayor que i elementos impares de la secuencia original, y que $i-1$ elementos pares (no $i+1$, pues los dos primeros impares no están precedidos de pares), por lo cual la posición de y en la secuencia mezclada es $> 2i-1$. Es decir, x e y deben ubicarse a partir de la posición $2i$ de A. Como los siguientes elementos, $P[i+1]$ e $I[i+2]$, deberán ubicarse a partir de la posición $2i+2$, considerando todas las posiciones, x e y sólo pueden posicionarse en $A[2i..2i+1]$.

El algoritmo $merge(1, n)$ procede entonces de la siguiente forma. Primero, cada procesador i se encarga de ubicar $A[i]$ en el lugar que le corresponde (reutilizamos A_1 para P y A_2 para I):

$$\textbf{if } pid \leq \left\lfloor \frac{n}{2} \right\rfloor \textbf{ then } b \leftarrow 0; t \leftarrow \left\lfloor \frac{n}{2} \right\rfloor \textbf{ else } b \leftarrow \left\lfloor \frac{n}{2} \right\rfloor; t \leftarrow n$$

$$\textbf{if } pid - b \bmod 2 = 0 \textbf{ then } d \leftarrow \left\lfloor \frac{b}{2} \right\rfloor + \left\lfloor \frac{pid-b}{2} \right\rfloor \textbf{ else } d \leftarrow b + \left\lfloor \frac{n-b}{2} \right\rfloor + \left\lceil \frac{pid-b}{2} \right\rceil$$

$$A[d] \leftarrow A[pid],$$

luego se invoca recursivamente como $merge(b, t)$, y finalmente $\frac{n}{2}$ procesadores se encargan cada uno de comparar dos celdas, $A[i]$ y $A[\lfloor \frac{n}{2} \rfloor + i + 1]$, y ubicarlas ordenadas en $A[2i..2i+1]$ (más un tratamiento especial para escribir $A[1] \leftarrow A[\lfloor \frac{n}{2} \rfloor + 1]$ y, si n es par, $A[n] \leftarrow A[\lfloor \frac{n}{2} \rfloor]$). La recursión termina cuando los arreglos son de tamaño 1, como siempre.

El algoritmo de mezclado resultante es EREW, de tiempo $T(n, n) = \log n$. El MergeSort consiste de $\log n$ invocaciones recursivas, cada una con su mezclado, por lo cual tiene $T(n, n) = \log^2 n$ y $E(n, n) = \frac{1}{\log n}$.

El algoritmo se puede correr con menos procesadores, pero la eficiencia no aumentará significativamente porque realiza trabajo $n \log^2 n$. Ejecutado con p procesadores, podemos

repartir la tarea de mezclado de manera que cada procesador se encargue primero de distribuir $\frac{n}{p}$ celdas en sus posiciones según su paridad, y a la vuelta de la recursión (cada una con $\frac{n}{2}$ elementos y $\frac{p}{2}$ procesadores) cada procesador se encargue de ubicar $\frac{n}{2p}$ pares en su posición final. La recursión termina cuando nos queda 1 procesador, en cuyo caso se mezcla secuencialmente en tiempo $\frac{n}{p}$. El tiempo total de mezclado es entonces $\frac{n}{p}\log p$. El ordenamiento completo tiene entonces tiempo $T(n,p) = \frac{n}{p}\log n \log p$, y la eficiencia será entonces $E(n,p) = \frac{1}{\log p}$.

8.7.2. Un algoritmo CREW con mejor eficiencia

Consideremos una forma distinta de mezclar. Cada celda $A_2[i]$ estará a cargo de un procesador. Éste realizará una búsqueda binaria de $A_2[i]$ en A_1 (las igualdades deben desempatarse en forma consistente, por ejemplo, poniendo antes a los elementos de A_1, lo que además nos da un ordenamiento estable). Digamos que la búsqueda binaria arroja que existen p elementos menores que $A_2[i]$ en A_1. Entonces podemos copiar $A_2[i]$ en su posición final, $A[i+p] \leftarrow A_2[i]$.

Estas $\frac{n}{2}$ búsquedas binarias ocurren en paralelo para todos los elementos de A_2, y se hacen otras $\frac{n}{2}$ búsquedas análogas en A_2, para los elementos de A_1. Luego de ellas, cada procesador escribe su celda en la posición final.

El resultado es un algoritmo CREW de mezclado que toma tiempo $T_M(n,n) = \log n$. El MergeSort completo requiere entonces tiempo $T(n,n) = T_M(n,n) \cdot \log n = \log^2 n$, y obtiene una eficiencia $E(n,n) = \frac{1}{\log n}$. A pesar de que el algoritmo realiza trabajo $n\log^2 n$, veremos que también podría realizar trabajo $n \log n$, de modo de obtener el mismo tiempo usando $\frac{n}{\log n}$ procesadores, y así mejorando la eficiencia.

Tomemos solamente $\frac{n}{2\log n}$ procesadores por un momento. Cada uno ahora estará a cargo de $\log n$ celdas consecutivas de A_2. Lo que hará será entonces buscar binariamente el último elemento de su zona en A_1. Al final, habremos obtenido las posiciones $p_1, p_2, \ldots, p_{n/(2\log n)}$ donde cada zona de A_2 termina en A_1. Por lo tanto, cada procesador i estará a cargo de mezclar su zona $A_2[(i-1)\frac{n}{2\log n} + 1..i\frac{n}{2\log n}]$, con la zona $A_1[p_{i-1}+1..p_i]$ (con $p_0 = 0$). El resultado de la mezcla se debe escribir a partir de $A[(i-1)\frac{n}{2\log n} + p_{i-1} + 1]$. Alguien debe también copiar la parte final $A_1[p_{n/(2\log n)} + 1..\frac{n}{2}]$ al final de A.

Todos los procesadores pueden entonces trabajar en paralelo haciendo su mezcla y escribiendo en la zona ya predeterminada de A, sin estorbarse. El problema con este esquema es que la partición de A_1 puede ser desbalanceada, tocándole a algunos procesadores más trabajo que a otros. En el peor caso, algún procesador puede trabajar sobre $\Theta(n)$ celdas y los demás deberán esperarlo.

La solución a esto es que los otros $\frac{n}{2\log n}$ procesadores particionen A_1 en forma regular, y busquen binariamente la posición que les corresponde en A_2, obteniendo las posiciones $q_1, q_2, \ldots, q_{n/(2\log n)}$. Si ahora *unimos* las posiciones regulares $p'_i = i\frac{n}{2\log n}$ con las irregulares p_i en A_1, obtendremos $\frac{n}{\log n}$ cortes p''_i en A_1 donde ningún intervalo es más largo que $\log n$. Si-

milarmente, unimos las posiciones regulares $q'_i = i\frac{n}{2\log n}$ con las irregulares q_i para particionar A_2 con $\frac{n}{\log n}$ cortes q''_i. Una vez realizadas las uniones, cada procesador i puede dedicarse a mezclar el intervalo $A_1[p''_{i-1}+1..p''_i]$ con $A_2[q''_{i-1}..q''_i]$, escribiendo a partir de $A[p''_{i-1}+q''_{i-1}+1]$, sin estorbarse y con la garantía de procesar a lo sumo $2\log n$ celdas entre los dos arreglos. Algún procesador debe también encargarse mezclar los dos intervalos finales de A_1 y A_2.

Nos falta resolver el subproblema de cómo unir los conjuntos ordenados p_i y p'_i, o q_i y q'_i. Note que estamos frente a un problema de mezclado similar al que estamos intentando resolver, con la diferencia de que los conjuntos suman $\frac{n}{\log n}$ elementos, por lo que tenemos un procesador por cada elemento y podemos mezclarlos usando nuestro método básico para cuando tenemos un procesador por elemento.

En total, tenemos un algoritmo CREW para ordenar con $T(n, \frac{n}{\log n}) = \log^2 n$ y $E(n, \frac{n}{\log n}) = 1$. No es difícil modificarlo para $p < \frac{n}{\log n}$ procesadores, obteniendo $T(n,p) = \log n(\frac{n}{p} + \log n)$ y eficiencia 1.

8.8. Máximo de un Arreglo en CRCW

Vimos que, en el modelo CRCW, se podía obtener el máximo de un arreglo en tiempo constante usando $\frac{n(n-1)}{2}$ procesadores. Diremos que este es el algoritmo de *dos pasos*. Tiene un tiempo muy bueno, pero su eficiencia es muy baja, $\frac{1}{n}$ ¿Es posible mejorar esa eficiencia?

Tratemos de ejecutar algo parecido a este algoritmo con n procesadores. Dividamos el arreglo en grupos de tamaño k, de manera que tengamos suficientes procesadores para darle $\frac{k(k-1)}{2}$ procesadores a cada grupo, así éste puede usar el algoritmo de dos pasos.

Al principio, haremos $\frac{n}{2}$ grupos de 2 elementos. Nos basta un procesador para determinar el máximo de cada grupo. Ahora nos quedan $\frac{n}{2}$ máximos, pero seguimos teniendo n procesadores. Esto nos permite definir grupos de tamaño 4, pues tendremos $\frac{n}{8}$ grupos en total y cada uno necesitará $\frac{4\cdot 3}{2} = 6$ procesadores, de modo que tenemos suficientes. Una vez encontrados estos $\frac{n}{8}$ máximos, podremos formar grupos de tamaño 16, pues tendremos $\frac{n}{8\cdot 16} = \frac{n}{128}$ grupos y necesitaremos $\frac{16\cdot 15}{2} = 120$ procesadores por grupo.

Más formalmente, en la iteración $\ell = 1$ tenemos n candidatos a máximo. El tamaño del grupo es $g_1 = 2$. Veamos que podemos tener $g_\ell = g_{\ell-1}^2 = 2^{2^{\ell-1}}$ en general. Al comenzar la iteración ℓ, tenemos

$$\frac{n}{g_1 \cdot g_2 \cdots g_{\ell-1}} = \frac{n}{2^1 \cdot 2^2 \cdot 2^4 \cdots 2^{2^{\ell-2}}} = \frac{n}{2^{1+2+4+\cdots+2^{\ell-2}}} = \frac{n}{2^{2^{\ell-1}-1}}$$

candidatos, por lo cual podemos formar grupos de tamaño $g_\ell = 2^{2^{\ell-1}}$, con lo que en total tendremos $\frac{n}{g_1 \cdot g_2 \cdots g_\ell} = \frac{n}{2^{2^\ell - 1}}$ grupos. Cada grupo necesitará $\frac{g_\ell(g_\ell-1)}{2} < 2^{2^\ell-1}$ procesadores. El número total de procesadores que necesitamos es entonces $< \frac{n}{2^{2^\ell-1}} \cdot 2^{2^\ell-1} = n$.

Es decir, el número de procesadores por grupo se puede ir elevando al cuadrado en cada iteración del algoritmo, por lo cual éste termina luego de $O(\log\log n)$ pasos. Por lo tanto,

conseguimos tiempo $T(n,n) = \log\log n$, con eficiencia $E(n,n) = \frac{1}{\log\log n}$, mucho mejor que $\frac{1}{n}$ dado que, usando muchos menos procesadores, tenemos sólo un leve incremento en el tiempo.

8.9. Ficha Resumen

- Modelo PRAM y medidas de eficiencia.
- Suma (u operadores asociativos) en arreglos: $T(n) = \log n$, $W(n) = n$ EREW.
- Parallel prefix: $T(n) = \log n$, $W(n) = n$ EREW.
- List ranking: $T(n) = \log n$, $W(n) = n \log n$ EREW.
- Tour euleriano: $T(n) = \log n$, $W(n) = n$ EREW.
- Ordenar: $T(n,n) = \log^2 n$ y $E(n,n) = \frac{1}{\log n}$ EREW.
- Ordenar: $T(n, \frac{n}{\log n}) = \log^2 n$ y $E(n, \frac{n}{\log n}) = 1$ CREW.
- Máximo o mínimo: $T(n,n) = \log\log n$, $E(n,n) = \frac{1}{\log\log n}$ CRCW común.

8.10. Material Suplementario

Manber [Man89, cap. 12] dedica un capítulo muy elegante a algoritmos en el modelo PRAM, en el que basamos la mayor parte de nuestro capítulo, exceptuando el MergeSort paralelo. Tiene un problema más que no cubrimos, sobre sumar dos números binarios en paralelo. Asimismo, explica el Lema de Brent con algo más de detalle. A partir de la sección 12.4 considera problemas en otros modelos de paralelismo, incluyendo un circuito que ordena en $T(n,n) = \log^2 n$ en base al cual presentamos nuestro algoritmo EREW. Sedgewick [Sed92, cap. 40] describe este mismo circuito.

Baase [Baa88, cap. 10] trata algoritmos PRAM también y discute unos pocos de los que vemos en el capítulo, en particular nuestro MergeSort CREW. También describe un algoritmo bastante más complicado para detectar las componentes conexas de un grafo $G(V,E)$ en tiempo $\log|V|$ usando $2(|V| + |E|)$ procesadores, en el modelo CRCW arbitrario. Al final incluye técnicas para mostrar cotas inferiores al tiempo $T(n)$ de algoritmos PRAM.

Para profundizar en algoritmos paralelos, una excelente fuente es JáJá [Jáj92], que incluye mucho más material que el resto de la bibliografía mencionada: modelo PRAM y sus costos, técnicas generales, algoritmos en listas y en árboles, búsqueda y ordenamiento, algoritmos en grafos generales y planares, en strings, algoritmos numéricos, algoritmos aleatorizados, y cotas inferiores. En particular, el libro describe un algoritmo de ordenamiento con $T(n) = \log n$ y $W(n) = n \log n$ en el modelo CREW, indicando que puede llevarse (en forma no trivial) a EREW.

Otras fuentes online de interés:

- legacydirs.umiacs.umd.edu/~vishkin/PUBLICATIONS/classnotes.pdf
- homes.cs.washington.edu/~arvind/cs424/notes/l2-6.pdf
- stanford.edu/~rezab/classes/cme323/S17/notes/lecture19/final_prep.pdf
- www.ida.liu.se/~chrke55/courses/MULTI/slides/theory2.pdf

Apéndice A

Conceptos Básicos

En este apéndice enunciamos resultados que deben ser conocidos para poder seguir este curso, y damos referencias donde puede encontrarse una descripción completa.

A.1. Análisis de Algoritmos

Notación O, Ω, o, ω, manipulación.
Teorema maestro y solución de recurrencias.
Definición de caso promedio y peor caso.

A.2. Técnicas Algorítmicas Básicas

A.2.1. Dividir y Reinar

ej búsqueda binaria, $O(\log n)$.

A.2.2. Algoritmos Avaros

A.2.3. Programación Dinámica

A.3. Árboles de Búsqueda

Operaciones: buscar, insertar, borrar, rangos, predecesor y sucesor.

A.3.1. Árboles binarios

$O(\log n)$ si los elementos vienen en permutaciones aleatorias.

A.3.2. Árboles AVL

$O(\log n)$.

A.3.3. Árboles 2-3

$O(\log n)$.

A.4. Hashing

Operaciones: buscar, insertar, borrar.
$O(1)$ promedio si la función se comporta como random

A.4.1. Hashing abierto

Linked list

A.4.2. Hashing cerrado

Rehashing, linear probing, factores de carga permitidos.

A.5. Ordenamiento

A.5.1. MergeSort

$O(n \log n)$.

A.5.2. QuickSort

$O(n \log n)$ promedio, pero más rápido.

A.6. Colas de Prioridad

Operaciones

A.6.1. Heaps

$O(\log n)$

A.6.2. HeapSort

$O(n \log n)$ e in-place.

A.7. Mediana de un Arreglo

Y generalización a k-ésimo.

A.7.1. QuickSelect

$O(n)$ promedio

A.7.2. Algoritmo lineal

$O(n)$ peor caso, constante involucrada

A.8. Árboles y Grafos

A.8.1. Recorrido en DFS

De un árbol, y de un grafo para obtener un árbol generador

A.8.2. Árbol cobertor mínimo

Algoritmos de Kruskal y Prim.

A.8.3. Caminos mínimos

Algoritmos de Dijskstra y Floyd.

Bibliografía

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[Baa88] S. Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 2nd edition, 1988.

[BB88] G. Brassard and P. Bratley. *Algorithmics, Theory and Practice*. Prentice-Hall, 1988.

[BEY98] A. Borodin and R. El-Yaniv. *On-line Computation and Competitive Analysis*. Cambridge University Press, 1998.

[CHL07] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[CR02] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2002.

[CT06] T. Cover and J. Thomas. *Elements of Information Theory*. Wiley, 2nd edition, 2006.

[dBCvKO08] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[DPV08] S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill, 2008.

[Jáj92] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[Knu98] D. E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[KT06] J. Kleinberg and E. Tardos. *Algorithm Design*. Pearson Education, 2006.

[Lev07] A. Levitin. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley, 2nd edition, 2007.

[LTCT05] R. C. T. Lee, S. S. Tseng, R. C. Chang, and Y. T. Tsai. *Introduction to the Design and Analysis of Algorithms: A Strategic Approach*. McGraw-Hill, 2005.

[Man89] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[MS08] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[MSS03] U. Meyer, P. Sanders, and J. Sibeyn. *Algorithms for Memory Hierarchies*. LNCS 2625. Springer, 2003.

[Nav16] G. Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

[Sam06] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[Sed92] R. Sedgewick. *Algorithms in C++*. Addison-Wesley, 1992.

[Vit08] J. S. Vitter. *Algorithms and Data Structures for External Memory*. Now Publishers, 2008.

[Wei95] M. A. Weiss. *Data Structures and Algorithm Analysis*. Benjamin/Cummings, 2nd edition, 1995.