1 More Hashing

(Continued from previous section) Remember from class that a hash function is a mapping $h : \{1, \ldots, U\} \to \{1, \ldots, m\}$. In most applications of hashing, you'll typically see $m < U$.

Hashing-based data structures are useful since they ideally allow for constant time operations (lookup, adding, and deletion), although collisions can change that if, for example, m is too large or you use a poorly chosen hash function.

Exercise. *True/False. If h is a hash function from $\{1, \cdots, U\}$ to $\{1, \cdots, m\}$ chosen uniformly at random from a universal hash family, $\mathcal{H}$, then for any $a \neq b \neq c \in \{1, \cdots, U\}$, we have $\Pr[h(a) = h(b) = h(c)] \leq \frac{1}{m}$:*

Solution.
True. We chose $h \in \mathcal{H}$ uniformly at random where $\mathcal{H}$ is a universal hash family, therefore:

$$\Pr[h(x) = h(y)] \leq \frac{1}{m} \qquad (\forall 1 \leq x, y \leq U)$$

With the above in mind, we can show that:

$$\begin{aligned} \Pr[h(a) = h(b) = h(c)] &= \Pr[h(a) = h(b) \text{ and } h(b) = h(c)] \\ &\leq \Pr[h(a) = h(b)] && \text{(one event is likelier than both)} \\ &\leq \frac{1}{m} && \text{(see above)} \end{aligned}$$

Exercise. *Follow-up. In fact, we can make the stronger claim that given the above, $\Pr[h(a) = h(b) = h(c)] \leq \frac{1}{m^2}$:*

Solution.
False. While it is true that $\Pr[h(a) = h(b)] \leq \frac{1}{m}$ and $\Pr[h(b) = h(c)] \leq \frac{1}{m}$, we know nothing about the independence of the events.

Exercise. *You are given an array X that consists of n elements. Your goal is to find a triple of elements (x_i, x_j, x_k) such that $x_i + x_j + x_k = 0$. First, give an efficient algorithm that uses hashing. Then, give a more space efficient algorithm that doesn't use hashing.*

Solution.
We do a perfect hashing of the elements in the array. This takes $O(n)$ time and $O(n)$ space. We then loop over all pairs of elements (there are $O(n^2)$ pairs). For each pair, (x_i, x_j), we look to see if $-(x_i + x_j)$ is in the hash table. Note, we do have to be careful to ensure that $i \neq j \neq k$. This can be done by storing a value in the hash table that represents which index hashed to that bucket. This is an $O(n^2)$ algorithm which uses $O(n^2)$ space to store the hash table.

An iterative approach can be used to solve this problem in $O(n^2)$ with no constant space. We first sort the array in $O(n \log n)$. We then find some index i. Then we set $j = 0$ and$k = n - 1$ (if $i = 0$ we can just start j at 1 instead, and likewise for $i = n - 1$. We then check the value of$x_i + x_j + x_k$. If it is equal to 0 we are done! If it is less than 0 we increment j (thereby increasing the value of our sum), otherwise we decrement k (thereby decreasing the value of our sum). For each value of i we are doing a linear amount of work, so overall this is $O(n^2)$.

2 Skip Lists

Recall that a skip list is a randomized data structure which solves the predecessor problem. The predecessor problem is to maintain a set of keys $k_1, k_2, \cdots, k_n$ subject to the following operations:

- **insert**(k): insert a new item into the database with key k
- **delete**(x): delete item x from the database (we assume x is a pointer to the item)
- **pred**(k): return the item k' in the database with k' as large as possible such that $k' \leq k$ (so in particular, if there is an item with key k in the database, we will find that item)

For simplicity, we assume that all keys k_i are distinct.

Exercise. *Describe how you would augment the skip-list data structure to deal with duplicate keys. We define each of **delete** and **pred** as operations which should perform their respective function on any of the possible matching elements.*

Solution.
You can deal with duplicate keys by replacing each key with a tuple (t, k_i) where t is the number of matching duplicate keys. There are multiple ways to deal with this problem.

Exercise (Random Access Skip Lists)**.** *Describe how to augment skip lists so that they support random access of items in $O(\log n)$ time. That is, you want to be able to find the i-th largest element in $O(logn)$. time.*

Solution.
Have each node track how many items it has between it and the previous node in its level at the bottom list. You can then use these counts to walk through the skip list in the normal way to find the i-th item.

Note that simply tracking the position of each node will lead to complications during insert and delete.

Exercise. *You are given a set of points of the form (x_i, y_i), you want to support queries of the form: what is the largest y-coordinate among points with $x \leq q$ for some query q.Describe how to augment skip lists to support these queries.*

Solution.
Key the items in the skip list by their x-coordinates. Have each node track the largest y among items whose x-coordinates are between it and the previous node in its level. As you walk through the list to search for the x-coordinate matching q, you will pass through nodes who, altogether, exactly cover the x range from $-\infty$ to q and thus the largest y among their ranges is the answer.

3 Random Walks

A random walk is an iterative process on a set of vertices V. In each step, you move from the current vertex v_0 to each $v \in V$ with some probability. The simplest version of a random walk is a one-dimensional random walk in which the vertices are the integers, you start at 0, and at each step you either move up one (with probability 1/2) or down one (with probability 1/2).

2-SAT: In lecture, we gave the following randomized algorithm for solving 2-SAT. Start with some truth assignment, say by setting all the variables to false. Find some clause that is not yet satisfied. Randomly choose one the variables in that clause, say by flipping a coin, and change its value. Continue this process, until either all clauses are satisfied or you get tired of flipping coins.

We used a random walk with a completely reflecting boundary at 0 to model our randomized solution to 2-SAT. Fix some solution S and keep track of the number of variables k consistent with the solution S. In each step, we either increase or decrease k by one. Using this model, we showed that the expected running time of our algorithm is $O(n^2)$.

Exercise. *This weekend, you decide to go to a casino and gamble. You start with k dollars, and you decide that if you ever have n dollars where $n \geq k$, you will take your winnings and go home (your winnings will be $\$(n-k)$. You play a game where it costs $\$1$ to gamble. You then have equal probabilities of either receiving $\$2$, or receiving nothing (and thereby losing $\$1$). What is the probability that you will lose all your money? (ie, why shouldn't you gamble?).*

Solution.
First, we calculate the probability that you win. Let $P(m)$ be the probability that you win if you currently have $\$m$.

We can then set up a recurrence relation.

$$\begin{aligned} P(0) &= 0 && \text{(You have no money, so you can't gamble)} \\ P(n) &= 1 && \text{(You've reached your goal, so you win)} \\ P(m) &= \frac{1}{2}P(m-1) + \frac{1}{2}P(m+1) && \text{(either you win or lose the current bet)} \end{aligned}$$

It seems like $P(m)$ is the average of $P(m-1)$ and $P(m+1)$ for all $0 < m < n$, which suggests a linear function of some sort.

Thus, the linear function that would take $P(0) = 0$ and $P(n) = 1$ is $P(m) = m/n$, which we can confirm to be a correct solution.

Therefore, if you start with $\$k$, the probability that you lose all your money is $1 - k/n$.

Exercise. *Now suppose that there are $n+1$ people in a circle numbered $0, 1, \ldots, n$. Person 0 starts with a bag of candy. At each step, the person with the bag of candy passes it either left or right with equal probability. The* last *person to receive the bag wins (and gets to keep all the candy). So if you were playing, then you would want to receive the bag only after everyone else has received the bag at least once. What is the probability that person i wins?*

Solution.
Define W_i as the event where person i wins. Then first note that $\Pr(W_0) = 0$ (the person who receives the bag first will never win!)

For all other W_i where $i \neq 0$, we can calculate $\Pr(W_i)$ by conditioning on two events. In order for W_i to occur, every other person must have received the bag at some point. Let us consider the two neighbors of i. Either $i - 1 \mod (n+1)$ (the person to the left of i) or $i + 1 \mod (n+1)$ (the person to my right) received the bag before the other (another way to think about this is that the bag must arrive from either i's left or right). Let F_l and F_r be the events where the left neighbor received it before the right and the right neighbor received it before the left. Then:

$$\Pr(W_i) = \Pr(W_i \mid F_l)\Pr(F_l) + \Pr(W_i \mid F_r)\Pr(F_r)$$

To calculate $\Pr(W_i \mid F_l)$, we consider the moment when $i - 1 \mod (n+1)$ receives the bag for the first time (this is before $i + 1 \mod (n+1)$ has received it). Then we have a configuration of the form $i, i+1, i+2, \ldots, n, 0, 1 \ldots, i-2, i-1$. The probability that i wins is therefore equivalent to the probability that the bag of candy makes it all the way to $i - 1$ without it ever being received by i (at that point, everyone except i has received the bag at least once and i will therefore win).

For this to occur, the bag of candy must walk $n - 1$ steps (has $\$(n-1)$ dollars). On each step, it can go backward or forward with equal probability (it either loses or gains \$1). i will win if the bag walks all $n-1$ steps (loses all $\$(n-1)$ dollars) without ever being received by i (without earning $\$n$ or more dollars).

By the previous problem, we have $\Pr(W_i \mid F_l) = 1 - \frac{n-1}{n} = \frac{1}{n}$.

By symmetry, $\Pr(W_i \mid F_r) = 1/n$ also.

Therefore:

$$\begin{aligned}\Pr(W_i) &= \Pr(W_i \mid F_l)\Pr(F_l) + \Pr(W_i \mid F_r)\Pr(F_r) \\ &= \frac{1}{n}\left(\Pr(F_l) + \Pr(F_r)\right) \\ &= \frac{1}{n}\end{aligned}$$

4 Short Min-Cut

For review, recall Karger's Algorithm: To try to find a minimum cut on a graph where all edges have weight 1, we randomly contract edges until there are only two vertices left, and then output the weight of the cut which separates the two vertices.

- To contract an edge (uv) means replacing (u, v) by a new node w, and then replacing u or v with w in all edges of the graph where they occur.
- With probability at least $\frac{1}{\binom{n}{2}}$, Karger's algorithm will output a particular minimum cut.