Warm-up

Problem 1. Check your understanding: summarise the key differences between a hash table and a Bloom filter, in terms of time and space complexity and guarantees provided.

Solution 1. Hash table: $O(\log m + m' \log m)$ v.s. Bloom filter: $O(k \log m + m')$ in space complexity (Typically, k is a constant.). When the number of buckets is large enough at order $m' = O(n)$.

Bloom filter does not actually store the elements, just the bits representing if they are in the set – that's why it could be wrong sometimes.

Hash table: $O(1)$ in expectation (e.g., for separate chaining) vs. Bloom filter: $O(1)$ worst case in LOOKUP and INSERT. However bloom filter can make mistakes sometime (false positives) and the simple version seen in class cannot handle REMOVE.

Problem 2. Prove the claim made in class: the expected time complexities of INSERT, LOOKUP, and REMOVE with separate chaining are all $O(1 + \alpha)$, where $\alpha = n/m'$ is the load of the hash table. What is their *worst-case* time complexity?

Solution 2. All of them depends on the number items in one bucket – in an expected worst case sense.

Over the randomisation of $h \sim \mathcal{H}$, after inserting $x_1, \ldots, x_{n-1}$, how many operations do you need to perform to insert x_n? Or look up one element after inserting $x_1, \ldots, x_{n-1}$? Or remove x_n from $x_1, \ldots, x_{n-1}$? They all depend on the size of the bucket $h(x_n)$. Denote $T(x_1, \ldots, x_n)$ as the number of operation one needs to perform for INSERT, LOOKUP or REMOVE.

$$\mathbb{E}_{h \sim \mathcal{H}}[T(x_1, \ldots, x_n)] = \mathbb{E}_{h \sim \mathcal{H}}[N_{h(x_n)}],$$

where $N_{h(x)}$ denote the size of the bucket for $h(x)$ after $x_1, \ldots, x_{n-1}$ is inserted over the randomisation of h.

By linearity of expectation. Given a universal hash family $\mathcal{H}$, we know for any $x \neq x'$, the following holds:

$$\Pr_{h \sim \mathcal{H}}[h(x) = h(x')] \leqslant \frac{1}{|\mathcal{Y}|}.$$

Without loss of generality, we will assume that $x_1, \ldots, x_{n-1}$ are distinct (as this is the hardest case). We can compute the expectation as follows:

$$\mathbb{E}[N_{h(x_n)}] = \mathbb{E}\left[\sum_{i=1}^{n-1} \mathbb{1}_{\{h(x_i) = h(x_n)\}}\right] = \sum_{i=1}^{n-1} \mathbb{E}[\mathbb{1}_{\{h(x_i) = h(x_n)\}}] = \sum_{i=1}^{n-1} \Pr_{h \sim \mathcal{H}}[h(x) = h(x')] \leqslant \frac{n-1}{|\mathcal{Y}|}.$$

And $|\mathcal{Y}| = m'$.

In the absolute worst case, there are at most $O(n)$ elements in any bucket.

Problem solving

Problem 3. Give an example of a universal hash family $\mathcal{H}$ from a universe $\mathcal{X}$ to a set $\mathcal{Y}$ for which the inequality is not always an equality:

$$\Pr_{h\sim\mathcal{H}}[\, h(x) = h(x') \,] \leq \frac{1}{|\mathcal{Y}|} \qquad \text{for all distinct } x, x' \in \mathcal{X}$$

Solution 3. Credits to someone in the tutorial session: consider the hash family $\mathcal{H} = \{h_1, h_2\}$ where $h_1(0) = 0, h_1(1) = 1$, and $h_2(0) = 1, h_2(1) = 0$.

For more, see, e.g., observations in `https://www.cs.purdue.edu/homes/hmaji/teaching/Fall%202017/lectures/14.pdf`.

Problem 4. Given three arrays A, B, and C each containing n positive integers, the task is to decide if there exist $1 \leq i, j, k \leq n$ such that $A[i] + B[j] = C[k]$. We aim for an algorithm running in (expected) time $O(n^2)$. *(We assume that, given a suitable hash function, we can evaluate it on any given input in constant time.)*

a) As a warm-up, describe an $O(n^3)$-time deterministic algorithm.

b) Describe an efficient $O(n^2)$ (expected) time algorithm.

c) Prove its correctness, and expected time complexity.

d) Analyze its worst-case time complexity. Can you get $O(n^2)$ here as well?

Solution 4.

a) The baseline algorithm is to iterate over all $1 \leq i, j, k \leq n$ triples (there are n^3 of them) and, for each of them, check if $A[i] + B[j] = C[k]$.

b) Consider the following algorithm: we create a hash table T, and insert all n elements from C in T. Once this is done, we loop over all n^2 possible pairs $1 \leq i, j \leq n$, and for each of them do a lookup in T to see if T contains the value $A[i] + B[j]$: if it does, we know there exists some k such that $A[i] + B[j] = C[k]$ and return true. If no such pair i, j is found, then we can return false.

c) Suppose there exist i^*, j^*, k^* such that $A[i^*] + B[j^*] = C[k^*]$. After inserting all element from C in T, the hash table contains the value $C[k^*]$; which means that, when looping over all pairs i, j, we will consider i^*, j^* and return true after performing a lookup for $A[i^*] + B[j^*]$ in T. Conversely, if the algorithm returns true at some iteration i, j, then this means T contains the value $A[i] + B[j]$; but since we inserted the prices listed in V (and only those values) into T, then there must be some index k such that $C[k] = A[i] + B[j]$.
In total, the algorithm performs n insertions into the hash table T and at most n^2 lookups. All options of collision handling mentioned in class (e.g., linear probing, separate chaining, and cuckoo hashing) have expected $O(1)$ insertions and lookups, so the total *expected* time complexity is $O(n) + O(n^2) = O(n^2)$.

d) We perform n insertions and at most n^2 lookups in the hash table. Depending on the choice of collision handling, this means the following worst-case time complexity:

- Using linear probing or chaining: all operations take $O(n)$ worst-case time. This means that the worst-case time complexity is $O(n^3)$.
- Using cuckoo hashing: insertions still takes $O(n)$ time in the worst case. However, now lookups are only $O(1)$ time even in the worst case, and so the total worst-case time complexity is $O(n^2)$.

Problem 5. Consider the following *two-level hashing* strategy: as in separate chaining, we will use a hash table A of size $m' = O(n)$ to contain our n items, and deal with collisions by having each of the m' buckets handle its hashed elements on its own. But instead of having a linked list for each bucket, we will instead use a secondary *hash table* for each bucket. Here we focus on the case where all n elements are inserted at once at the beginning, and we want to focus on the lookups.

a) Suppose that bucket k has n_k of the n elements hashed to it. What should be the size of the hash table A_k (the hash table in in bucket k) to guarantee it only has a collision with probability $1/2$?

b) Briefly describe how to do the batch insertion of all n elements (initialisation of the data structure).

c) Analyse the expected time complexity of a lookup to your hash table.

d) Analyse the expected space complexity of the overall data structure, and show it is $O(n)$.

Solution 5.

a) Suppose we make size of table m. The number of collision in expectation is.

$$\mathbb{E}\left[\#\text{number of collisions}\right] = \sum_{0<i<j<n_k} \mathbb{1}_{\{h(i)=h(j)\}} = \sum_{0<i<j<n_k} \Pr_{h\sim\mathcal{H}}\left[h(x) = h(x')\right] \leqslant \frac{\binom{n_k}{2}}{m}.$$

Set $m = 2\binom{n_k}{2} = O(n_k^2)$. By Markov's inequality,

$$\Pr\left[\#\text{number of collisions} \geqslant 1\right] \leqslant 1\mathbb{E}\left[\#\text{number of collisions}\right] \leqslant \frac{\binom{n_k}{2}}{m} \leqslant \frac{1}{2}.$$

b) Pick your first hash function h.

1. Hash all n elements and find out each n_k, for $k = 1, \ldots, m'$. Assuming $O(1)$ operation cost for hashing: $O(m') = O(n)$.
2. For the k-th position, initialise your secondary hash table with size $O(n_k^2)$. (If there is a collision, rehash until there isn't any. A constant number of rehashings is enough in expectation, and with high probability, for each fixed k.)

c) $O(1)$. Because no collision in the previous step.

d) Space complexity: how many buckets are in there? First we look at one particular position k,

$$n_k = \sum_x \mathbb{1}_{\{h(x)=k\}}$$

Remember that the first hash function $h : m \to m'$.
Linearity of expectation:

$$\begin{aligned}
\mathbb{E}\left[\sum_{k=1}^{m'} n_k^2\right] &= \sum_{k=1}^{m'} \mathbb{E}[n_k^2] \\
&= \sum_{k=1}^{m'} \mathbb{E}\left[\left(\sum_x \mathbb{1}_{\{h(x)=k\}}\right)^2\right] \\
&= \sum_{k=1}^{m'} \mathbb{E}\left[\left(\sum_x \mathbb{1}_{\{h(x)=k\}}\right)\left(\sum_y \mathbb{1}_{\{h(y)=k\}}\right)\right] \\
&= \sum_{k=1}^{m'} \mathbb{E}\left[\left(\sum_x \sum_y \mathbb{1}_{\{h(x)=k\}} \cdot \mathbb{1}_{\{h(y)=k\}}\right)\right] \\
&= \sum_{k=1}^{m'} \mathbb{E}\left[\left(\sum_x \mathbb{1}_{\{h(x)=k\}} \cdot \mathbb{1}_{\{h(x)=k\}}\right) + \sum_{x \neq y} \mathbb{1}_{\{h(x)=k\}} \cdot \mathbb{1}_{\{h(y)=k\}}\right] \\
&= \sum_{k=1}^{m'} \sum_x \mathbb{E}[\mathbb{1}_{\{h(x)=k\}} \cdot \mathbb{1}_{\{h(x)=k\}}] + \sum_{k=1}^{m'} \sum_{x \neq y} \mathbb{E}[\mathbb{1}_{\{h(x)=k\}} \cdot \mathbb{1}_{\{h(y)=k\}}]
\end{aligned}$$

It's one if and only if $h(x) = k$.

$$\mathbb{E}[\mathbb{1}_{\{h(x)=k\}} \cdot \mathbb{1}_{\{h(x)=k\}}] = \mathbb{E}[\mathbb{1}_{\{h(x)=k\}}] = \Pr[h(x) = k].$$

It's one if and only if $h(x) = k$ and $h(y) = k$.

$$\mathbb{E}[\mathbb{1}_{\{h(x)=k\}} \cdot \mathbb{1}_{\{h(y)=k\}}] = \Pr[h(x) = k, h(y) = k].$$

Swapping the sum over, we get

$$\begin{aligned}
\text{LHS} &= \sum_x \sum_{k=1}^{m'} \mathbb{E}[\mathbb{1}_{\{h(x)=k\}} \cdot \mathbb{1}_{\{h(x)=k\}}] + \sum_{x \neq y} \sum_{k=1}^{m'} \mathbb{E}[\mathbb{1}_{\{h(x)=k\}} \cdot \mathbb{1}_{\{h(y)=k\}}] \\
&= \sum_x \left(\sum_{k=1}^{m'} \Pr[h(x) = k]\right) + \sum_{x \neq y} \left(\sum_{k=1}^{m'} \Pr[h(x) = k, h(y) = k]\right) \\
&= \sum_x 1 + \sum_{x \neq y} \Pr[h(x) = h(y)] \leqslant n + \frac{n(n-1)}{2} \frac{1}{m'} = O(n).
\end{aligned}$$

See for instance Section 5.7: `https://jeffe.cs.illinois.edu/teaching/algorithms/notes/05-hashing.pdf`

Problem 6. We will analyse the error probability of the Bloom filter seen in class. We will focus on the error rate, that is, how frequently we would expect LOOKUP to make a mistake, "on average." In what follows, assume we inserted a dataset S of n elements into the Bloom filter. We will make the following (false, but convenient) assumption that we have truly random hash functions: the $(h_i(x))_{i,x}$ are fully independent across elements $x \in \mathcal{X}$ and hash functions $1 \leq i \leq k$, and $h_i(x)$ is uniformly distributed in $\{1, 2, \ldots, m'\}$ for every i and every x:

$$\forall i, x, y, \quad \Pr[h_i(x) = y] = \frac{1}{m'}$$

a) Fix any $1 \leq i \leq m'$. After inserting n elements into our Bloom filter, what is the probability p_i that the i-th bit of our array A is set to 1?
Let $B := \frac{m'}{n}$ be the average number of extra bits used per element. Using the approximation $1 + x \approx e^x$ (very accurate for small x), show that $p_i \approx 1 - e^{-k/B}$.

b) *Error rate:* What is the probability that, when calling LOOKUP(x) on a key which was *not* inserted (not part of the n keys from S), the value returned is yes?

c) Say you have a target per-element storage value B in mind: $B = 8$ bits. What is the number of hash functions k you should use to minimise the probability of error?

d) For the setting $B = 8$, and the choice of k above, what is the error rate you should expect?

e) Let's use $k = 6$ hash functions and explore the trade-off between space (parameter B) and error rate – we could decide to use more space than 8 bits per element. What is the expected error rate if you increase B to 12 bits? 16? 32?

Solution 6.

a) Since we made the assumption of truly uniform hashing, the probability that, for any fixed element x inserted, the i-th bit is *not* set to 1 by the j-th hash function is equal to $1 - 1/m'$. By independence, since we have k hash functions and n elements, the probability that the i-th bit is *not* set to 1 is equal to $(1 - 1/m')^{kn}$, and so

$$p_i = 1 - \left(1 - \frac{1}{m'}\right)^{kn} \approx 1 - e^{-\frac{nk}{m'}} = 1 - e^{-\frac{k}{B}}$$

b) For this to happen, we need *all* k bits $h_1(x), \ldots, h_k(x)$ to be set to 1. By the previous question and our independence assumption, this happens with probability

$$p_1 \times \cdots \times p_k = \left(1 - e^{-\frac{k}{B}}\right)^k$$

c) Either eyeball it on a plot, or use calculus (differentiate $\left(1-e^{-\frac{k}{8}}\right)^k$ with respect to k). You might want to use `https://www.wolframalpha.com/...` In detail: letting $f(x) = (1-e^{-x/8})^x$, we want to minimise f. Differentiating, you can check that

$$f'(x) = f(x)\left(\frac{x}{8}\cdot\frac{1}{e^{x/8}-1} + \ln\left(1-e^{-x/8}\right)\right)$$

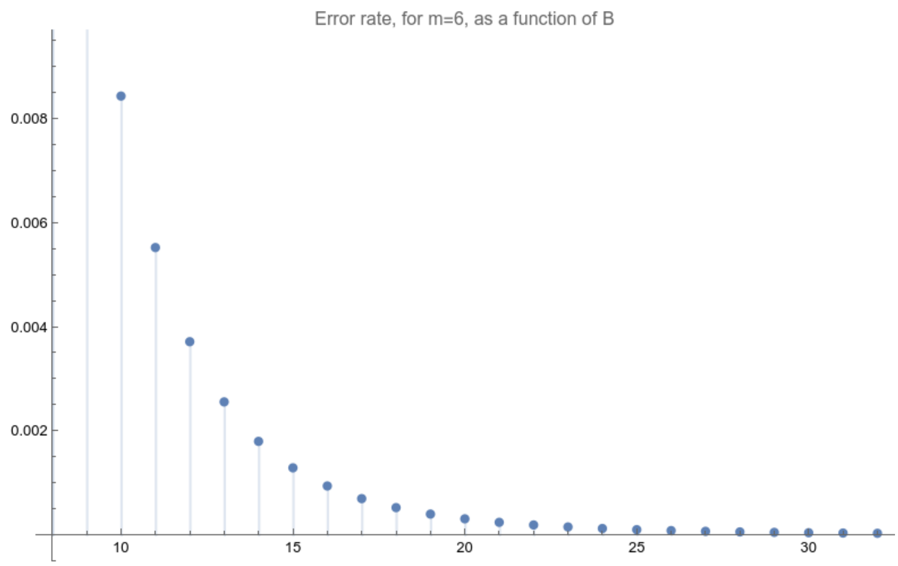
and, since $f(x) > 0$ for all $x > 0$, $f'(x) = 0$ if, and only if,

$$\frac{x}{8}\cdot\frac{1}{e^{x/8}-1} + \ln\left(1-e^{-x/8}\right) = 0.$$

Going further to argue that there is exactly one solution requires more calculus and is not very interesting, but you can check that plugging $x = 8\ln 2$ in the left-hand side does evaluate to 0: f is minimised for $x = 8\ln 2 \approx 5.6$.
The right answer is therefore $k = 6$ (the function is minimised for $k \approx 5.6$, and we need an integer). In general, one can derive the answer (again, based on the above approximations and assumptions, which are actually quite well supported in practice) to be $k = \lceil(\ln 2)B\rceil$. See, e.g., the above computation replacing 8 by B, or this computation on WolframAlpha.

d) We have $(1-e^{-6/8})^6 \approx 0.0216$, so the expected false positive rate when calling LOOKUP is roughly 2.16%.

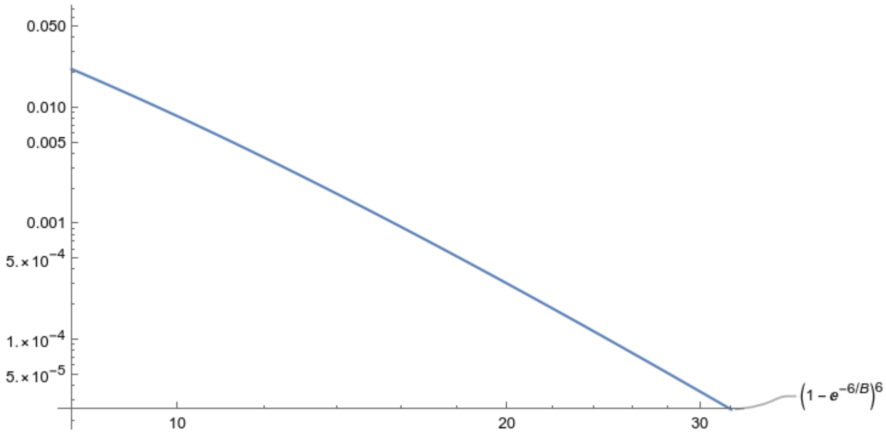
e) The corresponding values are 0.37%, 0.09%, and... 0.0025%.
The rate decreases quite fast as a function of B (for fixed k, n): see this plot:



Namely, the error rate decreases polynomially, roughly as

$$\Theta(1/B^6)\,.$$

Extra: why is the error rate $r(B)$ decreasing as $\Theta(1/B^6)$? One way to see it is to plot $\log r(B)$ as a function of $\log B$ (a "log log plot"), since if $r(B) = 1/B^c$ for some constant c, then $\log r(B) = \log(1/B^c) = -c \log B$ and the log log plot will look like a line with slope $-c$. Which is roughly what we observe here, for $c = 6$: Another



way is to see how the expression $r(B) = \left(1 - e^{-\frac{6}{B}}\right)^6$ from (d) behaves as B increases ($B \to \infty$): then $6/B \to 0$, and Taylor approximations ($e^u \approx 1 + u$ for small u) give us

$$\left(1 - e^{-\frac{6}{B}}\right)^6 \approx \left(1 - \left(1 - \frac{6}{B}\right)\right)^6 = \frac{6^6}{B^6} = \frac{46656}{B^6} = \Theta\left(\frac{1}{B^6}\right)$$

as claimed.

Advanced

Problem 7. Augment the Bloom filter data structure seen in class to add a Remove operation. Analyse the resulting guarantees (performance, error probability, space and time complexities).

Solution 7. *(Sketch)* One option is to use a secondary Bloom filter which keeps track of the deletions. (Note that this introduces a second type of errors now, false negatives, since the second Bloom filter has a small error probability of claiming an element was deleted.)

For a discussion, and other options, see, e.g., `https://cs.stackexchange.com/questions/19292/deleting-in-bloom-filters` (and references).