Randomized Algorithms and Data Structures

Zachary Yedidia

Introduction

A *randomized* algorithm is an algorithm that can make decisions based on the outcomes of random events ("coin flips"). Randomized algorithms come in two main types:

- **Las Vegas:** A Las Vegas algorithm is guaranteed to be correct, but the runtime is a random variable. With a Las Vegas algorithm we want to bound the expected runtime or show that it is small with high probability.
- **Monte Carlo:** A Monte Carlo algorithm has a fixed runtime but may give incorrect results with some probability. With Monte Carlo algorithms we want to achieve a low runtime as well as bound the probability that the algorithm outputs an incorrect answer.

Just as before, when examining the runtime of a randomized algorithm we look for worst-case guarantees. The algorithm's input is not random, even if its output is, and the input is assumed to be worst case as usual.

Probability review

Background in probability is assumed. A basic refresher is provided here.

Expectation: The expectation of a discrete random variable X with support $\mathcal{X}$ is

$$\mathbb{E}[X] = \sum_{x \in \mathcal{X}} x \mathbb{P}(X = x).$$

Linearity of expectation: For any $a, b \in \mathbb{R}$ and any random variables X, Y each supported on $\mathcal{S}$

$$\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y].$$

More generally, for any number of random variables X_i and constants a_i defined for $i = 1, \ldots, n$ we have

$$\mathbb{E}\left[\sum_{i=1}^{n} a_i X_i\right] = \sum_{i=1}^{n} a_i \mathbb{E}[X_i].$$

Markov's inequality: If X is a nonnegative random variable, then for any $\lambda > 0$

$$\mathbb{P}(X > \lambda \mathbb{E}[X]) < \frac{1}{\lambda}.$$

For example the probability that X takes on a value double its expectation is $\frac{1}{2}$.

Proof: Let $\mathcal{X}$ be the support of X, where every entry in $\mathcal{X}$ is nonnegative. Then

$$
\begin{aligned}
\mathbb{E}[X] &= \sum_{x \in \mathcal{X}} x\mathbb{P}(X = x) \\
&\geq \sum_{x \in \mathcal{X}, x > \lambda\mathbb{E}[X]} x\mathbb{P}(X = x) \\
&> \sum_{x \in \mathcal{X}, x > \lambda\mathbb{E}[X]} \lambda\mathbb{E}[X]\mathbb{P}(X = x) \\
&> \lambda\mathbb{E}[X]\mathbb{P}(X > \lambda\mathbb{E}[X]).
\end{aligned}
$$

Thus we have

$$\mathbb{E}[X] > \lambda\mathbb{E}[X]\mathbb{P}(X > \lambda\mathbb{E}[X])$$

which simplifies by dividing both sides by $\lambda\mathbb{E}[X]$ to

$$\frac{1}{\lambda} > \mathbb{P}(X > \lambda\mathbb{E}[X]).$$

Indicator random variables: An indicator random variable I_A for an event A is defined as

$$I_A = \begin{cases} 1 & \text{if } A \text{ happens} \\ 0 & \text{otherwise.} \end{cases}$$

We can think of I_A as a Bernoulli random variable where p is the probability that event A occurs. Therefore we can note

$$\mathbb{E}[I_A] = \mathbb{P}(A) = p.$$

Independence: Two events A, B are independent if knowing A does not provide any information about B. We can express this formally as

$$\mathbb{P}(A, B) = \mathbb{P}(A)\mathbb{P}(B).$$

For example, if a coin is flipped twice, the event that the first flip is heads is independent of the event that the second flip is heads: knowing that the first flip was heads does not give any information about what the second flip will be.

Similarly two random variables X, Y are independent if

$$\mathbb{P}(X = x, Y = y) = \mathbb{P}(X = x)\mathbb{P}(Y = y).$$

Exercise: An unfair coin comes up heads with probability $\frac{1}{3}$ and tails with probability $\frac{2}{3}$.

1. If the coin is flipped 3 times, what is the expected number of times it comes up as heads? How many heads should you expect if the coin is flipped 12 times?

> Let I_i be the indicator random variable for when the ith coin flip is heads. Note that each I_i is a Bernoulli random variable with $p = 1/3$ which is independent from the others. Then if N_3 is the number of heads in three coin flips, $N_3 = I_1 + I_2 + I_3$ and
>
> $$\mathbb{E}[N_3] = \mathbb{E}[I_1 + I_2 + I_3] = 3\mathbb{E}[I_1]3 \cdot \frac{1}{3} = 1.$$

We can do the same thing for 12 coin flips:

$$\mathbb{E}[N_{12}] = \mathbb{E}[I_1 + \ldots + I_{12}] = 12\mathbb{E}[I_1] = 12 \cdot \frac{1}{3} = 4.$$

2. On average, how many times should the coin be flipped in order for a heads to appear?

Let x be the number of times the coin should be flipped to get a heads. Then on the first flip there is a 1/3 chance of a heads, and a 2/3 chance that the coin will need to be flipped an additional x flips to get a head. Hence x has a recursive relationship with itself

$$x = \frac{1}{3} \cdot 1 + \frac{2}{3} \cdot (1 + x).$$

If we solve this for x we find $x = 3$, meaning we should expect a heads on the third flip.

Freivalds' algorithm

Suppose we have two matrices $A, B \in \mathbb{R}^{n \times n}$ that we would like to multiply. Using Strassen's algorithm (or some even better algorithms), we can achieve $o(n^3)$ time, but even the best known algorithms do not get to $O(n^2)$ (they are around $O(n^{2.7})$). Suppose that a cloud service company with much more powerful machines is willing to compute $C = AB$ on their machine and return C to us, and now we would like to verify that the C they gave us does equal AB.

Specifically in this problem we ask given 3 $n \times n$ matrices A, B, C is there a way to verify that $C = AB$ that is more efficient than directly computing AB.

The idea is to pick some $x \in \mathbb{R}^n$ and check that $Cx = ABx$. By associativity we can compute ABx as $A(Bx)$, which is a sequence of two vector multiplications which can be performed in $O(n^2)$ time. It is true that if $C = AB$ then $Cx = ABx$ but unfortunately in general it is possible that $Cx = ABx$ even though $C \neq AB$.

Freivalds' algorithm says to choose x *randomly*. We choose $x \in \{0, 1\}^n$ to be a random binary vector. In fact, we can generate k different random binary vectors $x^1, x^2, \ldots, x^k$ by selecting them independently and uniformly at random from $\{0, 1\}^n$. We then check if $Cx^i = ABx^i$ for $1 \leq k \leq k$ and if so we output that $C = AB$. If there is at least one i that this check fails for, we output that $C \neq AB$. However, it is possible for us to be fooled k times in a row and output a false positive. Thus Freivalds' algorithm is Monte Carlo with deterministic running time $\Theta(kn^2)$.

Algorithm 1 Freivalds' algorithm

```
1: procedure FREIVALD(C, A, B)
2:     for i = 1 to k do
3:         x ← RANDOM({0,1}^n)
4:         if Cx ≠ ABx then
5:             return false
6:     return true
```

Probability of correctness

Claim: We know that if $C = AB$ then $\mathbb{P}(Cx = Abx) = 1$. Now suppose $C \neq AB$. If $x \in \{0, 1\}^n$ is a binary vector chosen uniformly at random, then

$$\mathbb{P}(Cx \neq ABx) \geq 1/2$$

Proof: Let $D = C - AB$ so we want to show that $\mathbb{P}(Dx \neq 0)$. Since $C \neq AB$, D is not the zero matrix, and in particular there is some $1 \leq j \leq n$ such that D has a non-zero entry in column j. For any binary vector x, we define x' as being the vector obtained from flipping the jth bit in x (thus $x' = x + e_j$ or $x' = x - e_j$, where e_j is a vector with a one at index j and zeroes everywhere else). Now we claim that at least one of $Dx \neq 0$ and $Dx' \neq 0$ must be true, which is the same as saying that $Dx = Dx' = 0$ is impossible. Note that this would complete the proof as this would mean that for at least half of all vectors z, $Dz \neq 0$. So why is $Dx = Dx' = 0$ impossible? Suppose $Dx = 0$, then $Dx' = D(x \pm e_j) = Dx \pm De_j = \pm De_j$. But De_j is the jth column of D which we said was not zero.

Runtime

Claim: For any $0 < P < 1$, Freivalds' algorithm can be used to obtain an algorithm for the matrix multiplication verification problem with running time $\Theta(n^2 \log(1/P))$ such that if $C = AB$ it will be correct with probability 1 and if $C \neq AB$ then it will be correct with probability $1 - P$.

Proof: The runtime of Freivalds' algorithm is $\Theta(kn^2)$. The probability of failure if $C \neq AB$ is the probability that for each $x^1, \ldots, x^k$, $Cx^i = ABx^i$ is satisfied. Each of these events happens with probability 1/2, so the probability of this happening k times in a row is $P = 1/2^k$. Choose $k = \log(1/P)$ and the runtime will be $\Theta(n^2 \log(1/P))$ with a chance of failure of P (and chance of correctness of $1 - P$).

Randomized QuickSort

The goal of QuickSort is to sort n elements in an array $A[1 \ldots n]$. We will assume that each $A[i]$ is distinct (if they not we can use each element's index to break ties). The randomized QuickSort algorithm will be a Las Vegas algorithm with $\Theta(n \log n)$ expected running time.

The idea behind QuickSort is as follows: when given an array A, we pick a pivot element $A[p]$. We then compare $A[p]$ with $A[i]$ for every other i, creating two sets $S_L = \{i : A[i] < A[p]\}$ and $S_R = \{A[i] > A[p]\}$. We move the elements with indices in S_L to the left part of the array, followed by $A[p]$ followed by the elements with indices in S_R, and then recursively sort the left and right subarrays.

The question now is how do we pick the pivot element. In the worst case, we could pick the smallest element in the array each time, meaning we would have to recursively sort an array of size $n - 1$ at each iteration yielding

$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2).$$

In the best case we would pick the pivot to be the median element every time so that we sort two subarrays of size $n/2$ at each subcall, which would have runtime

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n).$$

In randomized QuickSort, we simply pick the pivot randomly. The idea is that on average the pivot element should split our recursions into two subarrays that are of size roughly $n/2$.

Algorithm 2 Randomized QuickSort

```
 1: procedure QuickSort(A, p, r)                          ▷ Sort a subarray A[p ... r]
 2:     if p < r then
 3:         q ← RandPartition(A, p, r)
 4:         QuickSort(A, p, q − 1)
 5:         QuickSort(A, q + 1, r)
 6: procedure RandPartition(A, p, r)        ▷ Rearrange the subarray A[p ... r] in place
 7:     i ← Random(p, r)
 8:     exchange A[r] with A[i]
 9:     x ← A[r]
10:     i ← p − 1
11:     for j = p to r − 1 do
12:         if A[j] ≤ x then
13:             i ← i + 1
14:             exchange A[i] with A[j]
15:     exchange A[i + 1] with A[r]
16:     return i + 1
```

Claim: The expected running time of QuickSort on an array of length n is $\Theta(n \log n)$.

Proof: For $1 \leq i < j \leq n$, let $X_{i,j}$ be an indicator random variable which is 1 if the ith smallest and jth smallest elements of A are *ever* compared over the course of QuickSort's execution, and let $X_{i,j} = 0$ otherwise. QuickSort's running time is proportional to the number of comparisons it makes, so the running time will be proportional to

$$\sum_{1 \leq i < j \leq n} X_{i,j}.$$

By linearity of expectation we know

$$\mathbb{E}\left[\sum_{1 \leq i < j \leq n} X_{i,j}\right] = \sum_{1 \leq i < j \leq n} \mathbb{E}[X_{i,j}] = \sum_{1 \leq i < j \leq n} \mathbb{P}(X_{i,j} = 1).$$

Now we want to know the probability that $X_{i,j} = 1$, or, in other words, the probability that i and j are compared at some point during the execution of QuickSort. At the start of the algorithm elements i and j are in the same subarray to be sorted. This continues as long as the pivot is chosen outside the interval $[i, j]$. At some point the pivot is chosen to be in the range $[i, j]$. If the pivot is i or j, then the two are compared. If not then i and j are never compared again because they are put in two separate subarrays. Since the pivot is chosen uniformly at random, the probability that the pivot is chosen to be i or j is $2/(j - i + 1)$. Thus, letting $k = j - i$, we have

$$
\begin{aligned}
\sum_{1\le i<j\le n} \mathbb{P}(X_{i,j}=1) &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j-1+1} \\
&= 2\sum_{i=1}\sum_{k=1}^{n-i} \frac{1}{k+1} \\
&\le 2\sum_{i=1}^{n} \int_{1}^{n-i+1} \frac{1}{k} dk \\
&= 2\sum_{i=1}^{n} \ln(n-i+1) \\
&\le 2\sum_{i=1}^{n} \ln n \\
&= 2n\ln n \\
&= \Theta(n\log n).
\end{aligned}
$$

QuickSelect

In the *selection* problem we are given an array $A[1\dots n]$ and an integer $1 \le k \le n$ and are asked to output the kth smallest value in the array. There is a deterministic algorithm that runs in $\Theta(n)$ time, but there were some complexities to make sure that the overall running time was linear. Here we present a similar but simpler algorithm that uses randomization.

In QuickSelect, we pick a pivot element x in A randomly, then partition the array into two sets (the elements less than x and those greater than x), then recursively call QuickSelect on the array containing the kth smallest element.

Algorithm 3 QuickSelect

```
1: procedure QuickSelect(A, p, r, k)          ▷ Returns the kth smallest element in A[p...r]
2:     if p = r then
3:         return A[p]
4:     if p < r then
5:         q ← RandPartition(A, p, r)
6:         if k = q then
7:             return A[k]
8:         else if k < q then
9:             return QuickSelect(A, p, q − 1, k)
10:        else
11:            return QuickSelect(A, p + 1, q, k)
```

We reuse the same partition function as in QuickSort.

Claim: The expected running time of QuickSelect on an array of length n is $O(n)$.

Proof: As before let's refer to the "ith item" as the ith smallest item in A. Let S_j be the set of items we recursively call QuickSelect on at recursive level j. Initially we call QuickSelect on all items, which is the interval of items $S_0 = \{1, \dots, n\}$. Call a recursive level j "good" if $|S_{j+1}| \le (3/4)|S_j|$. Let j_i be the index of the ith good level. Now let $n_0 = n$ and for $i > 0$ $n_i = |S_{j_i+1}|$. In other words n_i is the size of the subinterval after the ith good level. Then $n_i \le (3/4)n_{i-1}$ which implies $n_i \le (3/4)^i n$ by induction on i. Now we define the random variable $X_i = j_i - j_{i-1} + 1$. This is the number of recursive calls we go through after the $(i-1)$st good level before reaching the ith good level. Note that there are at most $d = \lceil \log_{4/3} n \rceil$ good levels, so the running time of QuickSelect is proportional to

$$|S_0| + |S_1| + \ldots \leq \sum_{i=0}^{d} X_i n_i \leq n \sum_{i=0}^{d} \left(\frac{3}{4}\right)^i X_i.$$

Thus the expected running time is at most

$$\mathbb{E}\left[n \sum_{i=0}^{d} \left(\frac{3}{4}\right)^i\right] = n \sum_{i=0}^{d} \left(\frac{3}{4}\right)^i \mathbb{E}[X_i].$$

We know that a level is guaranteed to be good as long as the pivot is not in the bottom or top quartile so, and thus any level is good with probability at least 1/2. We know for any random variable X supported on the natural numbers 0,1,2... that

$$\mathbb{E}[X] = \sum_{k=0}^{\infty} \mathbb{P}(X > k).$$

Then

$$\begin{aligned}\mathbb{E}[X_i] &\leq \sum_{k=0}^{\infty} \mathbb{P}(X_i > k) \\ &\leq \sum_{k=0}^{\infty} \frac{1}{2^k} \\ &= 2.\end{aligned}$$

Thus we can bound the running time of QuickSelect by

$$2n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i = 8n = O(n).$$

It is also possible to analyze the running time of QuickSelect using a similar method to the one we used for QuickSort. However, it becomes a little more messy with QuickSelect since we have to look at the three cases when k is less than i, between i and j, and greater than j, and we therefore need to solve 3 separate double summations like the one we solved for QuickSort.

Skip lists

We now examine the *dynamic predecessor problem.* The problem is to maintain a database, which is a set of keys $k_1, k_2, \ldots$, 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 (where x is a pointer to the item).
- Pred(k): return the item with key k' in the database with k' as large as possible such that $k' \leq k$ (note that if there is an item with key k, we will return that item).

Throughout this section, we let n be the number of items currently in the database. We also assume that all items in the database have distinct keys for simplicity (if not then the semantics of the database are not well-defined).

It is possible to support the three operations above in $O(\log n)$ worse-case time deterministically using data structures such as red-back trees, AVL trees, 2-3-4 trees, or other implementations of balanced

binary search trees. However, these data structures have quite complex mechanisms to remain balanced. Here we will see a randomized data structure called the *skip list* which is very simple and achieves $O(\log n)$ expected time for each operation.

We could solve the dynamic predecessor problem by simply using a sorted linked list. To insert a new value with key k we iterate until we find a key greater than k and insert it just before. Finding the predecessor is a similar story, and to delete a node we simply remove it from the linked this. Our runtime would be $O(n)$ for INSERT and PRED and $O(1)$ for DELETE.

One interesting modification we could make to this design would be to have two linked lists. The bottom list is the same one as before: a list of the sorted elements in the database. In addition, for every other element in the list, we add it to the top list, and let it keep a reference to the value in the bottom list that it corresponds to. For example, the figure below shows the data structure storing the keys $[1, 7, 8, 11, 18, 23, 25]$.

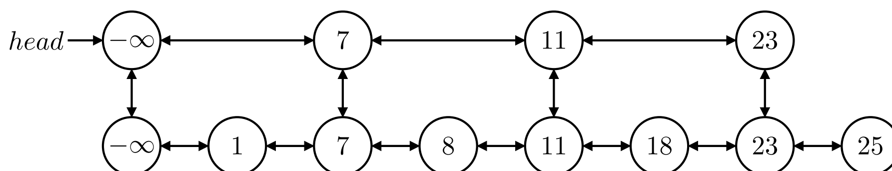

Figure 1: Skip to every second node

Now to find a certain key we can skip over every other node until a final visit at the end.

We could take this idea even further and add another layer with nodes corresponding to every fourth node in the database.

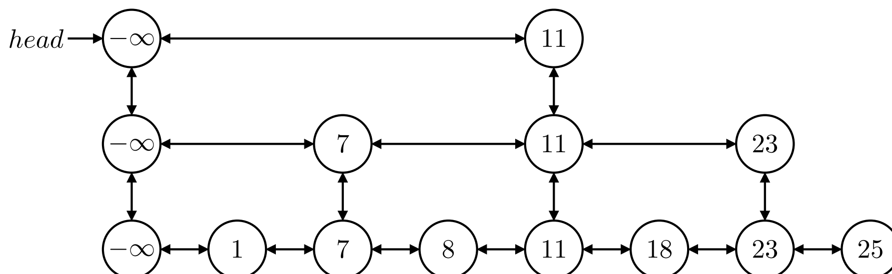

Figure 2: Skip to every second node and fourth node

Now if we need to find a node we can be even more efficient. To find 25 for example we can first skip to the 11 on the third level, then drop down one level and skip over to the 23 and finally drop down one level and move to the right to arrive at 25.

We could keep going with this idea such that we have a layer that skips every 2^ith node for $i, \ldots, \log n$. In fact this is great and very similar to a binary search in an array because finding a node takes $O(\log n)$ time. However, the work required to re-organize the data structure after a delete or insert operation is $O(n)$ because we have to maintain a strict order of node size. This is the same sort of complication that balanced binary trees solve, but they use complicated logic to do so.

The skip list works as follows. At the lowest level we keep all n items in a linked list in sorted order with $-\infty$ in the front. Then for each item we flip a fair coin to decide whether or not to promote it upward to the next layer. For each item that we promote, we flip a coin again to decide which ones to promote further. At some point no items will be promoted anymore and we'll be done constructing the data structure. We also always promote the $-\infty$ node and it appears at the beginning of each level. The figure below shows an example of how our previous array of keys might appear in a skip list.

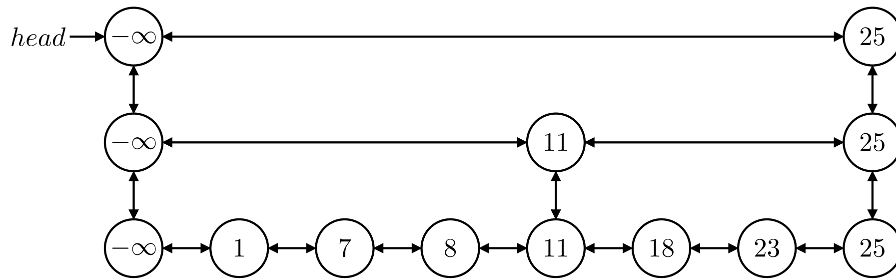

Figure 3: Possible configuration of a randomized skip list

To insert into the skip list, we insert into the bottom row and then promote the node randomly. Creating the list is the same as inserting all the elements starting with an empty skip list.

We now move to the analysis of the skip list.

Space consumption

Claim: The expected space consumption of a skip list is $O(n)$

Proof: Let the random variable X_i be the number of copies of item i in the data structure. Then the space consumption is proportional to $\sum_{i=1}^{n} X_i$. Then the expected space is proportional to

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i].$$

Now we know that X_i is the number of flips of a fair coin needed until seeing tails for the first time. Then we know that $\mathbb{E}[X_i] = 2$, and thus the expected space consumption is $O(2n) = O(n)$.

Running time

Claim: For any item $i \in \{1, \dots, n\}$ the expected time to query i is $O(\log n)$.

Proof: When we query for element i we start at the top left $-\infty$ and move right and down until we arrive at the lower level value for i. We make right moves unless the item to the right is bigger than i or doesn't exist, in which case we move down. The running time is therefore the number of right moves plus the number of down moves. We can only move down H times where H is the number of levels in the skip list. The number of right moves is the number of distinct items we touch while executing the query. Let Y_j be an indicator random variable which is 1 if we ever touch item j on query i and 0 otherwise. Thus the running time is proportional to

$$H + \sum_{j=1}^{i-1} Y_j.$$

Note that it is not possible to touch items larger than i in a query for i. Thus the expected running time is

$$\mathbb{E}[H] + \sum_{j=1}^{i-1} \mathbb{E}[Y_j].$$

Let us first analyze $\mathbb{E}[H]$. Since H is a nonnegative integer random variable we know

$$\begin{aligned}\mathbb{E}[H] &= \sum_{k=0}^{\infty} \mathbb{P}(H > k) \\ &= \sum_{k=0}^{\log n - 1} \mathbb{P}(H > k) + \sum_{k=\log n}^{\infty} \mathbb{P}(H > k).\end{aligned}$$

Since probabilities must be ≤ 1 we can replace the $\mathbb{P}$ in the first sum with 1. This gives

$$\mathbb{E}[H] \leq \log n + \sum_{k=\log n}^{\infty} \mathbb{P}(H > k).$$

Now let h_r denote the height of item r. The probability that the height of the tree is greater than k is the probability that there exists an item with height greater than k:

$$\begin{aligned}\mathbb{P}(H > k) &= \mathbb{P}(\exists r \in \{1, \ldots, n\} : h_r > k) \\ &\leq \sum_{r=1}^{n} \mathbb{P}(h_r > k).\end{aligned}$$

For $h_r > k$ to occur, we must have promoted r at least $k+1$ times, meaning $k+1$ coin flips were heads:

$$\begin{aligned}\mathbb{P}(H > k) &\leq \sum_{r=1}^{n} \frac{1}{2^{k+1}} \\ &= \frac{n}{2^{k+1}}.\end{aligned}$$

Now we can simplify our equation from before, letting $t = k + 1 - \log n$

$$\begin{aligned}\mathbb{E}[H] &\leq \log n + \sum_{k=\log n}^{\infty} \frac{n}{2^{k+1}} \\ &= \log n + \sum_{t=1}^{\infty} \frac{n}{2^{t+\log n}} \\ &= \log n + \sum_{t=1}^{\infty} \frac{1}{2^t} \\ &= \log n + 1.\end{aligned}$$

Now we bound $\mathbb{E}[Y_j]$. The item $j < i$ is touched while querying i iff $h_j = \max(h_j, h_{j+1}, \ldots, h_i)$. In other words j is touched if it is in a "top right" position on the path to i. Take a look at the figures of skip lists above to convince yourself. Note that item j may not achieve this maximum uniquely. As a result we have

$$\begin{aligned}\mathbb{E}[Y_j] &= \mathbb{P}(Y_j = 1) \\ &= \mathbb{P}(h_j = \max(h_j, h_{j+1}, \ldots, h_i)) \\ &= \mathbb{P}(h_j + 1 > \max(h_{j+1}, \ldots, h_i)).\end{aligned}$$

It then follows that for any fixed value $\alpha = \max(h_{j+1}, \ldots, h_i)$, $\mathbb{P}(h_j + 1 > \alpha) = \mathbb{P}(h_j > \alpha - 1)$. Then as we already saw above for $h_j > \alpha - 1$ to occur, we must have promoted j at least α times, meaning

$$\mathbb{P}(h_j > \alpha - 1) = \frac{1}{2^\alpha} = 2 \cdot \frac{1}{2^{\alpha+1}} = 2\mathbb{P}(h_j > \alpha).$$

This allows us to simplify the above expression for $\mathbb{E}[Y_j]$ to

$$\mathbb{E}[Y_j] = 2\mathbb{P}(h_j > \max(h_{j+1}, \ldots, h_i)).$$

Finally, conditioned on there being a unique maximum h_t over $t \in \{j, j+1, \ldots, i\}$ its index is equally likely to be anywhere in that interval, so we simplify to

$$\mathbb{E}[Y_j] \leq \frac{2}{i - j + 1}.$$

Therefore we have

$$\begin{aligned}
\mathbb{E}[H] + \sum_{j+1}^{i-1} \mathbb{E}[Y_j] &\leq \log n + 1 + 2\sum_{j=1}^{i-1} \frac{1}{i - j + 1} \\
&\leq \log n + 1 + 2\sum_{j=1}^{i-1} \frac{1}{i - j} \\
&= \log n + 1 + \sum_{k=1}^{i-1} \frac{1}{k} \\
&\leq \log n + 2 + \int_1^{i-1} \frac{1}{k} dk \\
&= \log n + 2 + \ln(i - 1) \\
&= O(\log n).
\end{aligned}$$

Exercise: In the above analysis we gave an upper bound for the expected runtime of a query on element i in a skip list with n elements using $p = \frac{1}{2}$ for the probability of promotion. What is a bound for the expected runtime using general p, and how can we choose p to optimize this bound?

In a similar vein to the above derivation we find $\mathbb{E}[H]$ and $\mathbb{E}[Y_j]$:

$$\begin{aligned}
\mathbb{E}[H] &\leq \log_{1/p}(n) + \sum_{k=\log_{1/p}(n)}^{\infty} P(H > k) \\
&\leq \log_{1/p}(n) + \sum_{\log_{1/p}(n)}^{\infty} np^{k+1} \\
&= \log_{1/p}(n) + \frac{p}{1 - p}
\end{aligned}$$

and

$$\mathbb{E}[Y_j] < \frac{1}{p} \cdot \frac{1}{i - j + 1}.$$

so then

$$\sum_{j=1}^{i-1} \mathbb{E}[Y_j] < \sum_{j=1}^{i-1} \frac{1}{p} \cdot \frac{1}{i-j+1} \\ \leq \frac{1}{p} \ln(i).$$

Then we can find an overall upper bound for the expected runtime of

$$B(i) = \log_{1/p}(n) + \frac{p}{1-p} + \frac{1}{p} \ln(i).$$

To find the optimal p we take the derivative and set to 0:

$$\frac{\partial B(i)}{\partial p} = \frac{\ln(n)}{p(\ln(p))^2} + \frac{1-2p}{(1-p)^2} - \frac{\ln(i)}{p^2} = 0.$$

This is difficult to solve for p analytically, so we would need to use numerical methods to estimate an optimal value for p.

Hashing

Dynamic dictionary problem

Consider the following data structural problem called the *dynamic dictionary problem*. We have a set of items each of which is a key-value pair. The keys are in the range $\{1, \ldots, U\}$ and the values are arbitrary. A data structure which supports the following operations is called a *dynamic dictionary*:

- INSERT(k, v): insert a new item into the database with key k and value v. If an item with key k already exists in the database, update its value to v.
- DELETE(x): delete item x from the database (we assume x is a pointer to the item).
- QUERY(k): return the value associated with key k, or *nil* if the key k is not in the database.

Note that the dynamic predecessor problem is strictly harder than this problem.

One randomized solution to the dictionary problem is *hashing*.

Definition: A hash family $\mathcal{H}$ is a set of functions

$$h : \{1, \ldots, U\} \to \{1, \ldots, m\}$$

where usually $U \gg m$.

Here U is the size of the universe and m is the size of the abbreviation we are assigning to each element of the universe. Consider for example hashing people based on birthdays. We would have $U = 7,000,000,000$ and $m = 365$ where each of the 7 billion people would have a birthday as one of the 365 days of the year. Even though our hash function is defined for all 7 billion people in the world, we only examine a subset of the population at once. We typically use n to denote the number of items we are going to hash.

Exercise: Suppose n items are hashed into m buckets with a perfectly random hash function (meaning that for any x, $h(x)$ is equally likely to be any of the m possible values).

1. What is the expected number of buckets that have exactly one item?

For a given bucket the chance that an item will land in that bucket when hashed is $\frac{1}{m}$. If we hash n items then the probability that *exactly* one item will land in any given bucket is a binomial distribution with 1 "success" and $n-1$ "failures:" $n\left(\frac{1}{m}\right)\left(1-\frac{1}{m}\right)^{n-1}$. Therefore we expect $mn\left(\frac{1}{m}\right)\left(1-\frac{1}{m}\right)^{n-1}$ buckets to have exactly one item.

2. What is the expected number of buckets that have exactly two items?

This is the same as the previous question except now we have 2 successes and $n-2$ failures, so we should expect $m\binom{n}{2}\left(\frac{1}{m}\right)^2\left(1-\frac{1}{m}\right)^{n-2}$ buckets to have exactly two items.

3. What is the expected number of buckets that have strictly more than two items?

For each bucket, we need to calculate the probability that it has more than 2 items hashed into it. This is the complement of the event that either 0, 1, or 2 items were hashed into it. The probability that no item landed in a particular bucket is $(1-\frac{1}{m})^n$ and we know from the previous two questions the probabilities for exactly 1 and 2 items. Thus the overall expectation is

$$m\left[1-\left(1-\frac{1}{m}\right)^n - n\left(\frac{1}{m}\right)\left(1-\frac{1}{m}\right)^{n-1} - \binom{n}{2}\left(\frac{1}{m}\right)^2\left(1-\frac{1}{m}\right)^{n-2}\right].$$

Hashing-based data structures are useful because they ideally allow for constant time operations for insert, delete and query. The idea of the solution is to maintain an array of size m where an item with key k is stored in the array in position $h(k)$. Unfortunately, since U is larger than m by the pigeonhole principle we may get *collisions* where two distinct keys $k \neq k'$ have $h(k) = h(k')$.

There are several ways to deal with collisions while hashing:

- **Chaining:** each bucket i holds a set of all items that are hashed to it such that $h(k) = i$. Adding an item just involves appending to the set, and querying involves iterating over the set.
- **Linear probing:** if bucket $h(k)$ already has an item, try $h(k)+1, h(k)+2, \ldots$ until an empty location is found (all taken mod m). If the table's size rises above a certain threshold then we replace the entire table by a new one that is larger by a constant factor.
- **Double hashing:** use two hash functions: $h(i,k) = h_1(k) + ih_2(k)$. If $h(0,k)$ is taken, try $h(1,k), h(2,k), \ldots$ until you find an empty location. This generalizes linear probing.

Of course, if the hash function is "bad" then the above schemes may not be efficient. For example, consider $\mathcal{H}$ only containing the hash function $h(i) = 1$. Then if we use chaining the entire data structure is a linked list. However, if $\mathcal{H}$ is "nice" in a certain way, and m is sufficiently large, we will be able to provide good guarantees.

Definition: A family $\mathcal{H}$ of hash functions mapping $\{1,\ldots,U\}$ into $\{1,\ldots,m\}$ is universal if for all $1 \leq x < y \leq U$,

$$\mathop{\mathbb{P}}_{h\in\mathcal{H}}(h(x) = h(y)) \leq \frac{1}{m}, \quad x \neq y$$

where h is chosen uniformly at random from $\mathcal{H}$. In other words, the probability of a collision for any $h \in \mathcal{H}$ is at most $\frac{1}{m}$.

Definition: A family $\mathcal{H}$ of hash functions mapping $\{1,\ldots,U\}$ into $\{1,\ldots,m\}$ is C-almost universal if for all $1 \leq x < y \leq U$,

$$\mathop{\mathbb{P}}_{h\in\mathcal{H}}(h(x) = h(y)) \leq \frac{C}{m}, \quad x \neq y$$

where h is chosen uniformly at random from $\mathcal{H}$. This is simply a generalization of the definition above.

Example: Pick some prime $p \geq U$ and define $h_a(x) = (ax \bmod p) \bmod m$. Then we let $\mathcal{H} = \{h_a : 0 < a < p\}$. We can then choose p to be at most polynomial in U so that any $h \in \mathcal{H}$ can be represented using $\log|\mathcal{H}| = O(\log U)$ bits. The analysis of this scheme requires some abstract algebra, but this family turns out to be almost universal, for some constant C that goes to 2 as $p \to \infty$.

Exercise: For some prime p, consider hashing n-digit strings consisting of letters from $\{0, 1, \ldots, p-1\}$ (so here, $U = p^n$) into the buckets $\{0, 1, \ldots, p-1\}$ (so $m = p$) as follows: randomly generate the n-tuple $(c_1, c_2, \ldots, c_n)$ by selecting each component randomly at uniform from $\{0, 1, \ldots, p-1\}$. Then for any key $\vec{x} = (x_1, x_2, \ldots, x_n)$, hash according to the function $h(\vec{x}) = \sum_{i=1}^n c_i x_i (\text{mod } p)$. Prove that this process forms a universal hash family.

> Consider two keys $\vec{x}$ and $\vec{y}$ such that $\vec{x} \neq \vec{y}$. Since they are not equal there is some index k such that $x_k \neq y_k$. In the process of generating $(c_1, c_2, \ldots, c_n)$ randomly, consider first generating all components except for c_k. Let $s_x = c_1x_1 + \ldots + c_{k-1}x_{k-1} + c_{k+1}x_{k+1} + \ldots + c_nx_n (\text{mod } p)$ (so the normal hash function without the c_kx_k term), and define s_y in the same way for $\vec{y}$. Then we have that $h(\vec{x}) = h(\vec{y})$ if and only if $s_x + c_kx_k = s_y + c_ky_k (\text{mod } p)$. If we solve this expression for c_k we get $c_k = (s_y - s_x)(x_k - y_k)^{-1} (\text{mod } p)$. Since there is a unique c_k that makes $h(\vec{x}) = h(\vec{y})$ and c_k is chosen uniformly from $\{0, \ldots, p-1\}$ there is a $\frac{1}{p} = \frac{1}{m}$ chance that $h(\vec{x}) = h(\vec{y})$, and thus this process forms a universal hash family.

Claim: Consider a hash table with chaining on a database with n items using a universal hash family $\mathcal{H}$ with $m \geq n$. Executing a query on key k takes expected time $O(1 + T)$ where T is the cost of evaluating a hash function $h \in \mathcal{H}$.

Proof: A query performs one hash evaluation, taking time T, followed by traversing the list at $A[h(k)]$. Let X_i be an indicator random variable that is 1 if the ith key k_i in the database collides with k (i.e. $h(k) = h(k_i)$) and 0 otherwise. The running time of a query is proportional to

$$T + \sum_{i=1}^{n} X_i.$$

Thus the expected running time is

$$T + \sum_{i=1}^{n} \mathbb{E}[X_i] = T + \sum_{i=1}^{n} \mathbb{P}(h(k) = h(k_i)) \leq T + \sum_{i=1}^{n} \frac{1}{m} = T + \frac{n}{m}.$$

Since $m \geq n$, this is $O(T + 1)$.

Bloom filters

A *bloom filter* is a probabilistic data structure used for set membership problems that is more space efficient than conventional hashing schemes. We have a list of m bits as well as k hash functions $f_1, \ldots, f_k$ each mapping from $\{1, \ldots, U\}$ to $\{1, \ldots, m\}$. When we want to add an element x into the set, we evaluate $f_1(x), \ldots, f_k(x)$ and set the corresponding bits in our array to 1. To check if an element x is in the set, simply check if the corresponding bits are set to 1.

Algorithm 4 Bloom Filter

```
1: procedure INSERT(b, x)                  ▷ Insert item x into the bloom filter b
2:     for i = 1 to k do
3:         b[f_i(x)] ← 1
4: procedure QUERY(b, x)                   ▷ Returns true if x is in the bloom filter b
5:     for i = 1 to k do
6:         if b[f_i(x)] ≠ 1 then
7:             return false
8:     return true
```

With bloom filters we trade away *correctness* for *space*. We know that if x is in the set and we are asked for it we will always return the correct answer, but if x is not in the set, we may incorrectly say that it is in the set. The bloom filter is clearly quite space efficient because we only store one bit per slot in the hash table.

Note that we cannot delete an element from a bloom filter. This is because some elements may share the same bit which flags their membership. If we set this bit to zero then we will have false negatives which we don't want to allow. One of the guarantees of the bloom filter is that there may only be false positives.

Claim: Consider a bloom filter with m bits, k perfectly random hash functions, and n elements already inserted. The chance of a false positive is approximately $\left(1 - e^{-kn/m}\right)^k$.

Proof: When inserting an item, the probability that a given bit is not set to 1 is $1 - \frac{1}{m}$. Then the probability that the bit is not set to 1 by any hash function is $\left(1 - \frac{1}{m}\right)^k$. Since we have inserted n items into the table already, the probability that a certain bit was not set to 1 by any of them is $\left(1 - \frac{1}{m}\right)^{kn}$. The probability that it is a 1 is the complement: $1 - \left(1 - \frac{1}{m}\right)^{kn}$. Now we assume that we are queried for some element x that is not in the set. The probability that all k bits corresponding to x are 1 is

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k$$

and this can be approximated by a Taylor expansion as

$$\left(1 - e^{-kn/m}\right)^k.$$

Counting bloom filters

A slight modification to the bloom filter, called the *counting bloom filter* allows deletions at the expense needing more space. Now instead of each slot having a single bit, they each have j bits (corresponding to a counter), and adding x to the set increments the counters at slots $f_1(x), \ldots, f_k(x)$, and deleting x from the set decrements the counters at those slots (after checking x's membership). This is similar to reference counting in garbage collection or file descriptor tables.

Algorithm 5 Counting Bloom Filter

```
 1: procedure Insert(b, x)                      ▷ Insert item x into the bloom filter
 2:     for i = 1 to k do
 3:         s ← f_i(x)
 4:         b[s] ← b[s] + 1
 5: procedure Query(b, x)                       ▷ Returns true if x is in the bloom filter
 6:     for i = 1 to k do
 7:         if b[f_i(x)] = 0 then
 8:             return false
 9:     return true
10: procedure Delete(b, x)                      ▷ Deletes x from the bloom filter
11:     if Query(b, x) then
12:         for i = 1 to k do
13:             s ← f_i(x)
14:             b[s] ← b[s] − 1
```

However, it is possible for these counters to overflow.

Exercise: What is the probability a counting bloom filter with k hash functions mapping into m j-bit counters overflows at a specific counter after adding n elements?

Each of the n elements is hashed k times and each of the nk hashes has probability $1/m$ to land on a certain counter. Let X be a random variable denoting how many times a specific counter is hashed to. The probability that a certain counter is hashed to i times out of the total nk hashes is given by the binomial distribution as

$$\binom{nk}{i}\left(\frac{1}{m}\right)^{i}\left(1-\frac{1}{m}\right)^{nk-i}.$$

For a j-bit counter, the max value is $2^j - 1$ so it overflows when $X \geq 2^j$, and thus the probability of overflowing at a specific counter is

$$1-\sum_{i=0}^{2^j-1}\mathbb{P}(X=i) = \boxed{1-\sum_{i=0}^{2^j-1}\binom{nk}{i}\left(\frac{1}{m}\right)^{i}\left(1-\frac{1}{m}\right)^{nk-i}.}$$

Sources

These notes are a combination of the course lecture notes, section notes, and my notes from lecture. The course was taught by Professor Jelani Nelson in Spring 2019. At various points I have also referred to the course textbook *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein.