# CS 161: Section 4 Solutions

## 1 A Different Universal Hash Family

Let's say we have $u$-bit long binary keys (so your universe is of size $2^u$), and we want to insert our keys into a hash table of size $M = 2^b$. We can number our buckets with $b$-bit long bit strings. For example, we might want to hash the key 10010 (so $u = 5$) and your hash table has positions $000, 001, 010, ..., 111$ (so $b = 3$). We talked in class about using tricks of modular arithmetic; here's a different way to form a universal hash family using matrix multiplication!

Because our keys and buckets are bit strings, we can think of them as vectors of bits. So our example key is now the length-5 vector $(1, 0, 0, 1, 0)$. How can we transform this length-5 vector into a length-3 vector? One natural thing to do is to multiply by a $3 \times 5$ matrix of 1's and 0's, using binary (i.e., modulo 2) arithmetic. So to pick a hash function $h$, we pick a matrix $A$ and then let $h(x) = Ax$. Let's say we happen to pick the matrix:

$$A = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Using our example $A$ and our example vector as $x$, this gives us:

$$h(x) = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

1. Prove that this gives a universal hash family.

   **Solution.**

   First, let's recall how this vector-matrix multiplication works. If I want the $j$th entry in the *output* vector, I must sum $A[j, 0]x[0] + A[j, 1]x[1] + \cdots + A[j, u]x[u]$. (Where $A[0, j]$ is the entry in the $j$th row, $i$th column, and $u$ is the width of the matrix.) Do out an example if you feel shaky on this, and convince yourself that this is correct!

   To prove that this construction gives us a universal hash family, we need to show that if we randomly chose a hash function $h$, then $\Pr[h(x) = h(y)] \leq 1/n$ for every $x \neq y$ in our universe. We can think of this as having an arbitrary fixed $x$ and $y$, and then picking a random hash function and checking whether $h(x) = h(y)$.

   If $x \neq y$, then in particular there must exist some index, $i$, where $x[i] \neq y[i]$. Without loss of generality, let's say that $x[i] = 0$ and $y[i] = 1$. Now, let's imagine randomly building up our matrix, $A$. Let's say we build all the columns except for the $i$th column. Note that this means, even before we know the $i$th column, we know what $Ax$ is! That's because $x[i] = 0$, so no matter what $A[j, i]$ equals, $A[i, j] \cdot x[i] = 0$. Moreover, we don't know what $Ay$ is at all! For example, if we're asking about the $j$th value in $Ay$, we know that it's determined by $A[j, 1]y[1] + A[j, 2]y[2] + \cdots + A[j, i]y[i] + \cdots + A[j, n]y[n]$. But $A[j, i]y[i] = A[j, i]$. since $y[i] = 1$! And because we're doing binary arithmetic, even if we know

all the other values in the sum, the value of $A[j,i]$ can completely determine the output. This means that there are $2^b$ different possible outputs for $Ay$, even if we know exactly what $Ax$ is! Only one of these outputs will give us $Ay = Ax$, and each one is equally likely. So for given $x$ and $y$, and randomly chosen $A$, we have just shown that $\Pr[Ax = Ay] \leq 1/2^b$. But $h(x) = Ax$, $h(y) = Ay$, and $n = 2^b$ is the size of the hash table! So this is saying that $\Pr[h(x) = h(y)] \leq 1/n$, which is exactly what we were trying to prove.

2. What is the size of this hash family? How does that compare with the family discussed in class? What might be the pros and cons of each hash family in a practical application?

   **Solution.**

   The size of this hash family is the number of possible binary matrices of size $u \times b$. Since there are $u \cdot b$ entries and each can have a 0 or 1, this gives us $2^{u \cdot b}$ total possible hash functions.

   To construct a hash function in the family discussed in class, once we'd picked $p$, we had to pick $a, b \in \{0, ..., p-1\}$. So, given $p$, there are $p^2$ possible hash functions. But, we needed to pick $p \geq |U|$, so to compare the size with the size of the hash family discussed above we need to note that $p \geq |U| = 2^u$. So our new hash family has size $\geq 2^{2u}$. Since $b \leq u$ above, this means that we actually get a larger hash family with this second construction.

   There's no single correct answer to which might be better in practice. Some factors to consider: the number of different functions in the family, the ease of computing the hash of a value, and the space necessary to store the hash function.

# 2 A Randomly Built BST

In this problem, we prove that the average depth of a node in a randomly built binary search tree with $n$ nodes is $O(\log n)$. A *randomly built binary search tree* with $n$ nodes is one that arises from inserting the $n$ keys in random order into an initially empty tree, where each of the $n!$ permutations of the input keys is equally likely.

Let $d(x, T)$ be the depth of node $x$ in a binary tree $T$ (the depth of the root is 0). Then, the average depth of a node in a binary tree $T$ with $n$ nodes is

$$\frac{1}{n} \sum_{x \in T} d(x, T) \ .$$

1. Let the *total path length* $P(T)$ of a binary tree $T$ be defined as the sum, over all nodes $x$ in $T$, of the depth of node $x$. We note that the average depth of a node in $T$ with $n$ nodes is equal to $\frac{1}{n}P(T)$. Show that $P(T) = P(T_L) + P(T_R) + n - 1$, where $T_L$ and $T_R$ are the left and right subtrees of $T$, respectively.

   **Solution.**

   Let $r(T)$ denote the root of tree $T$. Note the depth of node $x$ in $T$ is equal to the length of the path from $r(T)$ to $x$. Hence, $P(T) = \sum_{x \in T} d(x, T)$.

   For each node $x$ in $T_L$, the path from $r(T)$ to $x$ consists of the edge $(r(T), r(T_L))$ and the path from $r(T_L)$ to $x$. The same reasoning applies for nodes $x$ in $T_R$. Equivalently, we have

$$d(x, T) = \begin{cases} 0, & \text{if } x = r(T) \\ 1 + d(x, T_L), & \text{if } x \in T_L \\ 1 + d(x, T_R), & \text{if } x \in T_R \end{cases} .$$

Then,

$$\begin{aligned}\sum_{x\in T} d(x,T) &= d(r(T),T) + \sum_{x\in T_L} d(x,T) + \sum_{x\in T_R} d(x,T) \\ &= 0 + \sum_{x\in T_L} [1 + d(x,T_L)] + \sum_{x\in T_R} [1 + d(x,T_R)] \\ &= |T_L| + |T_R| + \sum_{x\in T_L} d(x,T_L) + \sum_{x\in T_R} d(x,T_R) \\ &= n - 1 + P(T_L) + P(T_R) \ .\end{aligned}$$

It follows that $P(T) = P(T_L) + P(T_R) + n - 1$.

2. Let $P(n)$ be the expected total path length of a randomly built binary search tree with $n$ nodes. Show that $P(n) = \frac{1}{n}\sum_{i=0}^{n-1} (P(i) + P(n-i-1) + n - 1)$.

**Solution.**

Let $T$ be a randomly built binary search tree with $n$ nodes. Without loss of generality, we assume the $n$ keys are $\{1, \ldots, n\}$.

By definition, $P(n) = \mathbf{E}_T[P(T)]$. Then, $P(n) = \mathbf{E}_T[P(T_L) + P(T_R) + n - 1] = n - 1 + \mathbf{E}_T[P(T_L)] + \mathbf{E}_T[P(T_R)]$, where $T_L$ and $T_R$ are the left and right subtrees of $T$, respectively. Note

$$\mathbf{E}_T[P(T_L)] = \sum_{i=1}^{n} \mathbf{E}_T[P(T_L)|r(T) = i] \cdot \Pr(r(T) = i) \ .$$

Since each element is equally likely to be the root of $T$, $\Pr(r(T) = i) = \frac{1}{n}$ for all $i$. Conditioned on the event that element $i$ is the root, $T_L$ is a randomly built binary search tree on the first $i - 1$ elements. To see this, assume we picked element $i$ to be the root. From the point of view of the left subtree, the elements $1, \ldots, i-1$ are inserted into the subtree in a random order, since these elements are inserted into $T$ in a random order and subsequently go into $T_L$ in the same relative order. Hence, $\mathbf{E}_T[P(T_L)|r(T) = i] = P(i-1)$. Putting these together, we get

$$\mathbf{E}_T[P(T_L)] = \sum_{i=1}^{n} \frac{1}{n} P(i-1) \ .$$

Similarly, we get $\mathbf{E}_T[P(T_R)] = \sum_{i=1}^{n} \frac{1}{n} P(n-i)$. Then,

$$\begin{aligned}P(n) &= n - 1 + \mathbf{E}_T[P(T_L)] + \mathbf{E}_T[P(T_R)] \\ &= n - 1 + \sum_{i=1}^{n} [P(i-1) + P(n-i)] \\ &= n - 1 + \sum_{i=0}^{n-1} [P(i) + P(n-i-1)] \ ,\end{aligned}$$

where we changed the indexing of the summation in the last equality.

3. Show that $P(n) = O(n \log n)$. You may cite a result previously proven in the context of other topics covered in class.

**Solution.**

This is the same recurrence that appears in the analysis of Quicksort in the notes from lecture 5.

4. Design a sorting algorithm based on randomly building a binary search tree. Show that its (expected) running time is $O(n \log n)$. Assume that a random permutation of $n$ keys can be generated in time $O(n)$

**Solution.**

The algorithm is 1) construct a randomly built binary search tree $T$ by inserting given elements in a random order; and 2) do the inorder traversal on $T$ to get a sorted list. Note step 2 can be done in $O(n)$ time. We argue that step 1 takes $O(n \log n)$ time in expectation. We observe that given the final state of tree $T$, we can compute the amount of work spent to construct $T$. To insert a node $x$ at depth $d$, we traversed exactly the path from the root to the parent of $x$, at depth $d-1$, to insert it. Hence, we can upper bound the total work done to construct $T$ by $O(P(T))$. From part (c), we know that $P(T) = O(n \log n)$ in expectation. It follows that Step 1 takes $O(n \log n)$ time in expectation. Overall, the algorithm runs in $O(n \log n)$ in expectation.