Readings

- [Lecture Notes Chapter 23: Hashing](#)

Problems

**Problem 1**

With \( n \) distinct balls in \( m \) distinct bins what is the probability that no bucket has more than 1 ball? You may assume that \( n \leq m \).

**Solution**

We will treat this as a counting problem. That is, we count the number of ways the event that no bucket has more than 1 ball, and divide by the total amount of ways to divide the \( n \) balls into \( m \) buckets. This technique works because each division of the \( n \) balls is equally likely. The numerator of our expression is \( mP_n = \frac{m!}{(m-n)!} \). The intuition behind this is that we have to pick exactly \( n \) bins out of our \( m \) bins if each bin that is filled has exactly one ball in it. Because our balls are distinct, we deal with permutations and not combinations. The denominator is immediately \( m^n \), because there are \( m \) choices for each of our \( n \) balls. This gives:

\[
\frac{m!}{(m-n)!}
\]

**Alternate Explanation/Solution**

Think about assigning each ball a label corresponding to which bin it is in. The total number of ways to put \( n \) balls in \( m \) bins is the total number of assignments, which is just \( m^n \) (\( m \) possible choices for each of the \( n \) balls). So that’s the denominator. Now, the probability that each bucket has at most one ball is the probability that each ball gets a unique label. To do this, assign ball 1 any of the \( m \) labels. Then assign ball 2 any of the \( (m-1) \) remaining labels. Then assign ball 3 any of the \( (m-2) \) remaining labels. etc. Since there are \( n \) distinct balls, this can be done in \( m \times (m-1) \times (m-2) \times ... \times (m-(n-1)) = m!/(m-n)! \) different ways. That’s the numerator.

**Problem 2**

Assume we have a hash table \( T \) of size 10 that uses linear probing and has hash function \( h(x) = x \mod 10 \). We insert 6 numbers into \( T \) and we get the below table:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>52</td>
</tr>
<tr>
<td>6</td>
<td>46</td>
</tr>
<tr>
<td>7</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
What is one possible order that we could have inserted these elements to get this result? How many probes would be required for inserting 13 in the table?

Solution

One possible order that the numbers could have been inserted in is 46, 34, 42, 23, 52, 33. For a sequence to be valid:

- 46 and 52 must be inserted before 33
- 42, 23, and 34 must be inserted before 33 and 52

It would take 6 probes to insert 13 into the table, as we would first try to insert it in index 3, fail, and then traverse the cluster until we reach index 8, at which point it would be inserted after we realize that $T[8]$ is empty. Note that this is a prime demonstration of primary clustering.

Problem 3

Design an algorithm that determines if two lowercase words are anagrams of each other in expected $O(n)$ time. **Note:** A string $A$ is an anagram of another string $B$ if $A$ is a permutation of $B$. Can you do it in worst case $O(n)$ time?

Solution

**Expected $O(n)$:** First of all, if $A$ and $B$ aren’t the same length, they cannot be permutations. Otherwise, use two hash tables where the keys are letters and the values are the counts of those letters. Iterate through $A$ incrementing the count of a letter in $A$’s hash table every time that letter is seen in $A$. Similarly, iterate through $B$ incrementing the count of a letter in $B$’s hash table every time that letter is seen in $B$. This effectively builds a map that maps from every letter in $A$ to the number of times that letter appears in $A$. It also does the same for $B$. Then, iterate through the letters in $A$’s map, and see if their counts are the same as their counts in $B$’s map. If this passes, we know $A$ and $B$ are the same. Note that we didn’t have to iterate through the letters of $B$ because we know $A$ and $B$ are the same length.

**Worst Case $O(n)$:** Because there are only 26 lowercase letters in the alphabet, we can just use 2 arrays of size 26 each to store the counts. Then, it’s just the same algorithm as was used above except the keys above are just indices in the arrays and the values above are the values in the arrays at those indices. Because arrays support worst case $O(1)$ time insertions and look-ups, and we iterate through each string once, this implementation runs in worst case $O(n)$ time.

Alternatively, you could just build the hash table/array the same way for string $A$, then traverse string $B$ and for each letter, decrease the corresponding counter in $A$’s map/array. If a counter ever becomes $−1$ or the letter did not exist in $A$, return false. Otherwise, return true (assuming we already checked if $A$ and $B$ are the same length).

Problem 4

You are given an array $A$ containing distinct randomly assorted integers. Your goal is to find two elements in the array whose sum is $k$ in $O(n)$ expected time.

Solution

Begin by traversing $A$ and putting all of the elements in a hash table. Then traverse $A$ again, and for each element $e$, check and see if $k − e$ exists in the hash table (and make sure $k − e ≠ e$). If it does, simply return $e$ and $k − e$. If you reach the end of the array and haven’t found a pair, return that no such pair exists.

Running time Analysis: It takes expected $O(n)$ time to insert all of the elements of $A$ into the hash table, $O(n)$ time to traverse $A$ again, and expected $O(1)$ time for each of the $n$ elements (yielding expected $O(n)$ time) to search for its complement. Therefore our algorithm still runs in expected $O(n)$ time.
Problem 5

How would you detect a cycle in a linked list of distinct elements in expected $O(n)$ time? Can you do it in constant space?

Solution

Algorithm: Traverse the linked list, and maintain a hash table of every element we’ve encountered. As we reach a new element in the linked list, check to see if it already exists in our hash table. If it does, we’ve found a cycle. If we reach the end of the linked list, then we know that no cycle exists.

Runtime Analysis: It takes linear time to traverse the linked list and expected $O(1)$ time for each lookup and insertion in the hash table, of which there could be $2n$. Thus, the algorithm is expected $O(n)$ time.

To do it in constant space, we can begin with two pointers $i$ and $j$ and traverse the array. To traverse the array, we move the pointer $j$ two spots along the linked list for every spot we move $i$. If there is a cycle, we know that $j$ will eventually intersect with $i$ (think about why this is!). If there is no cycle, then $j$ will reach the end of the linked list and we are done.

Note: We can say this algorithm is $O(n)$—try to figure out why this is and see if you can come up with a more specific bound in the case of a cycle!