SIG Algorithm Challenges

Week 3: Hash Tables

Dane's Contact Email: dziema2@uic.edu

Join our Slack channel, #sig_algorithm_chal on the UIC ACM slack

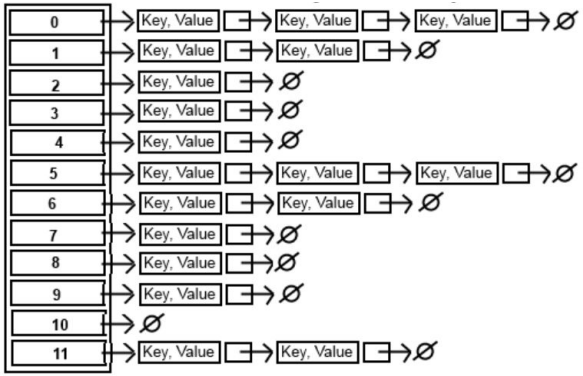
Hash Table Basics

Hash tables operate by inputting the key in a key value pair into a hash function to get an index into the hash table to store the key value pair

Can think of it like an array where the hash function tells you the index of a list to find the item

Example: a (string, integer) hash table of size 16 with hash function h, working with value (“dog”, 3)

- Insertion: Calculate h(“dog”) to get a number between 0-15, put (“dog”,3) in the end of that.
  - If hash table is too full (generally between ⅔ -¾ full) after this insertion, resize it (double the size)
  - Load factor: number of elements in table divided by table size
- Find: Calculate h(“dog”) to get a number between 0-15, search for dog in that list
  - May end up having to loop through whole list to find (“dog,3)

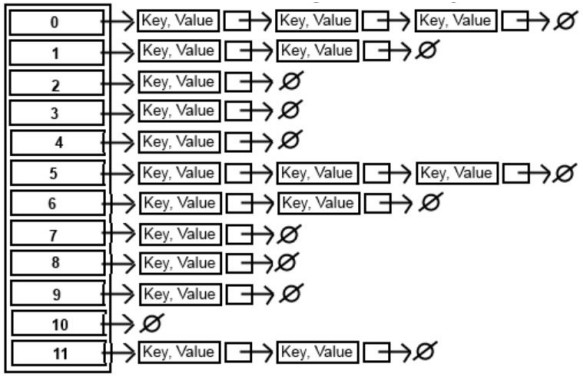


Hash Table Run Time

Hash tables are famous for their O(1) insert and find run time, but it's a little more complex than that

- Evaluation of hash function: O(1) time
  - Permutes a fixed number of bits
- Insert: O(1) to add an item to the end of the list, however, we have to resize the table if it gets too full.
  - Every time we resize the table, we have to rehash all the pairs, costing O(N) time
  - If we employ a size-doubling strategy, results in **O(1) amortized**
- Find: constant time to apply hash function, but can be O(N) to search through the list (if all the values in the table are in the same bucket)
  - With use of a uniform hash function, and table resizing, the buckets will on average only have a constant number of items (think 5-6, some small number not related to table size)
  - This gives **O(1) average case** run time

Moral: Hash tables are O(1) performance for find/insert on average



Hash Function Design

A hash function will map the domain of the keys to an index into the hash table

- Example: if our keys are strings like “dog”, “cat”, “bird”, then we would need a hash function h that goes from the domain of strings to the domain of integers

h: String -> Integer

In general, hash function design is a job best left for people who specifically study hash function However, it is useful to be able to identify the properties of a good has function. The following are some properties of a good hash function

- Fast to compute
  - Hash function is computed every time you perform a hash table operation, therefore it must be fast!
- Uniform Distribution
  - We want the buckets to contain very few values each
- Collision resistance
  - Collisions are inevitable, however we want as few of them as possible

For cryptographic hash functions, there is another property called “Preimage resistance”, which means it is hard to guess the input to a hash function given the resulting value

Likelihood of Collisions

Collisions are inevitable with a hash function, there are many more possible keys than there are possible indexes into the hash table. Collisions are more common than you would think originally

- Birthday paradox example: There are 366 possible birthdays, but you only need 20 people to have a > 50 % chance of a “birthday collision”
  - This is because there are 190 (20 choose 2) unique pairs of people in a group of 20, so it is a 190/366 chance that two people share a birthday if all birthdays are equally likely

- Corollary to hash tables: if we have 2,450 entries in our hash table of size 1,000,000 there is a > 95% chance of a collision

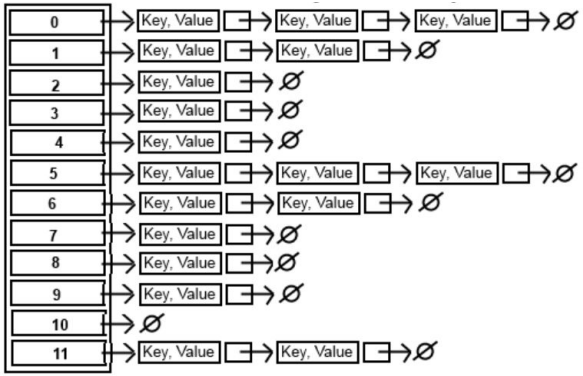
- Collision resolution strategy is necessary for a good hash table
  - The method of using a lists of lists to resolve collisions (the example in this presentation) is called “separate chaining”
  - Other methods such as linear probing or quadratic probing also exist, however they do not perform as well

Aside: Hashsets

One data structure that is used similarly to a hash table is a hashset

- A hashset is essentially the same thing as a hash table, except instead of storing a (key, value) pair, it stores a key only
  - Another way to think of it: the key and value are always the same

- Hashsets use less memory



Some Hash table problems can be solved with a hashset and use less memory

Identifying Hash Table Problems

Hash tables problems are extremely common. It is generally worth it to at least spend a minute or so to ask yourself “Can a hash table help me here?” at the start of an interview question

Some common hash table problem archetypes

- Find duplicates in a collection of objects

- Store a relationship of some kind between two entities
  - Example: some sort of rule to transform one type of item to another type
  - In Dynamic Programming, often times a hashtable is used to store and (input, output) pair from some function

- Count the number of occurrences of some event

General Tips for Hash Tables

- Know how to use a hash table in your chosen language
  - Java: HashMap<>, HashSet<>.
  - C++: unordered_map<>
  - Python {key1:value1, key2:value2}

- Make sure to familiarize yourself with the syntax of the hash tables in your language
  - The documentation is your friend!
  - Java: put(key, value), get(key), containsKey(key)
    - To iterate over a HashMap in java, use the .keySet() operator and iterate over that
    - E.g .for map with string keys do for (String key: map.keySet())
  - C++: []operator, find()
    - E.g. map[“dog”] = 3. If dog doesn’t exist in the table, it makes a new entry for it, otherwise it replaces the old one with 3
    - Can also do something like int x = map[“dog”] however, if the entry for dog doesn’t exist you may get unexpected behavior
    - Use the .find operator to see if a map contains a key, the find function returns an iterator to the element or map.end() if not found
    - Iterating over a HashMap in c++ is the same as any other collection (e.g. vector)

Limitations of Hash Tables

Hash tables problems are extremely powerful, perhaps the most useful of all data structures. However, they do have their limitations

- Hash Tables are unordered
  - if you are storing data that needs to be kept in a particular order then something like a balanced binary tree or priority queue may perform better

- Hash Tables don’t handle data with duplicates in them especially well
  - Can get around this with data processing, whether or not this is a good idea depends on your data

- Some datasets lead to a higher amount of collisions than normal
  - Can be solved by using a specially designed hash function for your data set

Sample Hash Table Problem

In a array `A` of size `2N`, there are `N+1` unique elements, and exactly one of these elements is repeated N times.

Return the element repeated `N` times.

Example 1:

```
Input: [1,2,3,3]
Output: 3
```

Example 2:

```
Input: [2,1,2,5,3,2]
Output: 2
```

Source: https://leetcode.com/problems/n-repeated-element-in-size-2n-array/

Problem Approach

Ideas for how to finish this problem

1: Go through the array and use a hashset to keep track of what numbers we have seen

2: If we encounter an element that is already in our hashset, that is the one that occurs multiple time

Pseudocode for this problem

```
repeatedElement(Array):
        For every element in the array
                If this element is in the hashset
                        Return the element
                Else
                        Add the element to the hash table
```

Java Code for this problem

```
class Solution {
    public int repeatedNTimes(int[] A) {
        //map that stores the count of each number
        HashSet<Integer> numbers = new HashSet<Integer>();
        for (int i=0; i<A.length; i++) {
            //if we have already seen a number then it must be the repeated one
            if (numbers.contains(A[i])) {
                return A[i];
            }
            numbers.add(A[i]);
        }
        return 0;
    }
}
```

Aside: Hash Table counter example

We have a list of points on the plane. Find the K closest points to the origin (0, 0).

Example:

```
Input: points = [[3,3],[5,-1],[-2,4]], K = 2
Output: [[3,3],[-2,4]]
```

Idea: Use a hashmap with (Key:distances, Values:index of that point)

1: store all distances, and their index in the array in hash table

2: Go through this hash table and select the K smallest distances

Problem: Distances may not be unique, and HashMaps are unsorted, so it isn't easy to get the K smallest

Better Data Structure: Priority Queue (Or TreeMap if distances are unique)

Rapid Fire Examples

Find the minimum length word from a given dictionary `words`, which has all the letters from the string `licensePlate`. Such a word is said to *complete* the given string `licensePlate`

Example:

```
Input: licensePlate = "1s3 PSt",

words = ["step", "steps", "stripe", "stepple"]

Output: "steps"
```

Use a hashmap with (Key:characters in license plate, Values:number of occurrences of that letter)

1: store all the characters, and their counts, in the license plate into a hash table

2: Create a copy of this hash table, for every word, keep decrementing the count of the letters when you encounter them. If all keys are 0, then the string completes the license plate

Rapid Fire Examples

We are given two sentences `A` and `B`. (A *sentence* is a string of space separated words. Each *word*consists only of lowercase letters.)

A word is *uncommon* if it appears exactly once in one of the sentences, and does not appear in the other sentence.

Example 1:

```
Input: A = "this apple is sweet", B = "this apple is sour"
Output: ["sweet","sour"]
```

Use a hashmap with (Key:Words in each string, Values:number of occurrences of that word)

1: Make two hash tables, one for each string, and store all the words and their counts in the map

2: Go through both hash tables, see if they have a count of

Rapid Fire Examples

In an alien language, surprisingly they also use english lowercase letters, but possibly in a different order. The order of the alphabet is some permutation of lowercase letters.

Given a sequence of words written in the alien language, and the order of the alphabet, return true if and only if the given words are sorted lexicographicaly in this alien language.

```
Input: words = ["hello","leetcode"], order = "hlabcdefgijkmnopqrstuvwxyz"
Output: true
Explanation: As 'h' comes before 'l' in this language, then the sequence is sorted.
```

Use a hashmap with (Key:English characters, Values:Alien Alphabetical position) (e.g., h:1, l:2 … in example above)

1: Add all the letters and their ordering into a hashmap

2: Go through each string, and see if it sorted relative to its neighbor using the hashmap

Rapid Fire Examples

Given a list of directory info including directory path, and all the files with contents in this directory, you need to find out all the groups of duplicate files in the file system in terms of their paths.

```
Input:
["root/a 1.txt(abcd) 2.txt(efgh)", "root/c 3.txt(abcd)", "root/c/d 4.txt(efgh)", "root 4.txt(efgh)"]
Output:
[["root/a/2.txt","root/c/d/4.txt","root/4.txt"],["root/a/1.txt","root/c/3.txt"]]
```

Use a hashmap with (Key: Contents of File, Values: First File found with those contents

1: insert the file contents and their file paths into a hashmap

2: if we find content that is already in the hashmap, add it to our duplicate list

HashMap Problems to Work On

Links to level 2-4 can be found at https://github.com/dane8373/SIG_Algorithm_Challenges/tree/master/Week3

Level 1) You're given strings J representing the types of stones that are jewels, and S representing the stones you have. Each character in S is a type of stone you have. You want to know how many of the stones you have are also jewels.

Level 2) Given an array of integers, return indices of the two numbers such that they add up to a specific target (this problem + its variations are in my opinion the most common hash table interview question)

Level 3) Given a string, sort it in decreasing order based on the frequency of characters. (example, “tree” becomes “eert” or “eetr”, because e appears twice, and the others only appear once, so e must come before r and t)

Level 4) Given a string, find the length of the longest substring without repeating characters.

Problem Approach for Level 1 problem

Can use a hashset for this problem

1: Load all of the jewel types into a hashset

2: For every character in the string, see if the hashset contains that character

3: If it does, increase the count of jewels by one, otherwise don’t

Java Code for level 1 problem

```
class Solution {
    public int numJewelsInStones(String J, String S) {
        int count = 0;
        //add all the jewels to a hashset
        HashSet<Character> jewels = new HashSet<Character>();
        for (int i=0; i<J.length(); i++) {
            jewels.add(J.charAt(i));
        }
        //go through all our rocks
        for (int i=0; i<S.length(); i++) {
            //increase count by 1 if a given rock is a jewel
            count += jewels.contains(S.charAt(i)) ? 1 : 0;
        }
        return count;
    }
}
```

Problem Approach for Level 2 problem

Observation 1: For every number i, there is exactly one other number j such that i+j = k. (j = k - i)

Create a HashMap of (Key:Value of array, Value: Index in array)

1: Go through the array and add all (vaue, index) pairs into the hash map

2: For every value i in the hashmap, see if k-i is in the map, if it is, then return the index of i and the index of k-i

3: If not, continue until such a pair is found

Java Code for level 2 problem (part 1)

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        //map that stores all the numbers with their indecies
        HashMap<Integer,Integer> map = new HashMap<Integer,Integer>();
        for (int i=0; i<nums.length; i++) {
            //if we found two of the same number, and that happens to sum up
            //to k, then we need to stop now and return those indecies
            if (map.containsKey(nums[i]) && 2*nums[i] == target) {
                int[] ret = new int[2];
                ret[0] = map.get(nums[i]);
                ret[1] = i;
                return ret;
            }
            map.put(nums[i],i);
        }
```

Java Code for level 2 problem (part 2)

```
        //go through everything in the map
        for (int key: map.keySet()) {
            //look for the number we need to add to this number to get K
            int other = target - key;
            //if we find it return the indecies
            if (map.containsKey(other)
                && map.get(key) != map.get(other)) {
                int[] ret = new int[2];
                ret[0] = map.get(key);
                ret[1] = map.get(other);
                return ret;
            }
        }
        return null;
    }
}
```

Problem Approach for Level 3 problem

Observation 1: We can accomplish this by counting the number of each letter, then rebuilding the string using the counts. (e.g. if we count 1 t, 1 r, 2 e's, then we know we need 2 e's first, then 1 r, then 1 t)

Create a HashMap of (Key: Character, Value: Count of that character)

1: Go through the array and add all (Character, count) pairs into the hash map

Problem: This hashmap is unordered, we can't look at which count is highest easily

Solution: Use the data from this hashmap with a different, ordered data structure (ordered by count)

2: For every value i in the hashmap, add the pair (count, character) into a new, ordered data structure (TreeMap, priorityQueue)

3: Build the string using this data structure

Java Code for level 3 problem (Part 1)

```
class Solution {
    public String frequencySort(String s) {
        //step 1: count the frequency of all elements with a hashmap
        HashMap<Character,Integer> counts = new HashMap<Character,Integer>();
        for (int i=0; i<s.length(); i++) {
            int count = 1;
            if (counts.containsKey(s.charAt(i))) {
                count += counts.get(s.charAt(i));
            }
            counts.put(s.charAt(i), count);
        }
```

Java Code for level 3 problem (Part 2)

```
//step 2: build an ordered map of Integer Character pairs ordered by the counts from the hashmap
//since there can be more than one character with the same count, we need to use an
//integer, ArrayList<character> pair, so we can store all unique pairs.
//count probably use a priority queue for this and be a little faster but I am more familiar with Treemaps
TreeMap<Integer,ArrayList<Character>> order = new TreeMap<Integer,ArrayList<Character>>();
for (Character key: counts.keySet()) {
    if (!order.containsKey(counts.get(key))) {
        order.put(counts.get(key),new ArrayList<Character>());
    }
    ArrayList<Character> temp = order.get(counts.get(key));
    temp.add(key);
    order.put(counts.get(key), temp);
}
```

Java Code for level 3 problem (Part 3)

```
        //Step 3: from smallest to biggest add the characters to the end of the string
        //rebuild the original string's letters by putting in the same count of each letter
        String ret = "";
        //for each frequency
        for (int i: order.navigableKeySet()) {
            ArrayList<Character> chars = order.get(i);
            //for each character at this frequency
            for (int k=0; k<chars.size(); k++) {
                //put in the same count of each letter
                for (int z=0; z<i; z++) {
                    ret = chars.get(k) + ret;
                }
            }
        }
        return ret;
    }
}
```

Problem Approach for Level 3 problem

Observation 1: We can accomplish this by counting the number of each letter, then rebuilding the string using the counts. (e.g. if we count 1 t, 1 r, 2 e's, then we know we need 2 e's first, then 1 r, then 1 t)

Create a HashMap of (Key: Character, Value: Count of that character)

1: Go through the array and add all (Character, count) pairs into the hash map

Problem: This hashmap is unordered, we can't look at which count is highest easily

Solution: Use the data from this hashmap with a different, ordered data structure (ordered by count)

2: For every value i in the hashmap, add the pair (count, character) into a new, ordered data structure (TreeMap, priorityQueue)

3: Build the string using this data structure

Problem Approach for Level 4 problem

Observation 1: If we find character c at indexes i and j, then we can eliminate all strings starting before i and ending later than j

Create a HashMap of (Key: Character, Value: Index of that character) and store the index of the last time we saw two consecutive characters (the first index of the two)

1: Starting from the front, for every character c, see if the character (Character, Index) pair is already in the hash map,

2: If it is, and it is at a point later than the last time we saw two consecutive elements, record this length if it is longer than our max and then we update index of the last time we saw two consecutive characters.

3: If it is not in the table, or the value is less than the index of the last time we saw two consectuive characters, put (Character, index) into the hasmap

4: Return the longest length we went without seeing two consecutive characters

Java Code for level 4 problem (Part 1)

```java
class Solution {
    public int lengthOfLongestSubstring(String s) {
        //map to store the latest index of a particular character
        HashMap<Character,Integer> map = new HashMap<Character, Integer>();
        int max = 0;
        //last index where we had to trim due to encountering a repeated character
        int lastDelete = 0;
        for (int i=0; i<s.length(); i++) {
            //if we encounter a character that is the same as one we have seen before
            //and we didn't already trim it off
            if (map.containsKey(s.charAt(i)) && map.get(s.charAt(i)) >= lastDelete) {
                if (i - lastDelete > max) {
                    max = i - lastDelete;
                }
                //update the trim point
                lastDelete = map.get(s.charAt(i)) + 1;
            }
            //update the index of this character
            map.put(s.charAt(i), i);
        }
```

Java Code for level 4 problem (Part 2)

```
        //if we never found a repeat then we just return the length
        if (lastDelete == 0) {
            return s.length();
        }
        //otherwise return the length minus the index of where we last trimmed
        if (s.length() - lastDelete > max) {
            max = s.length() - lastDelete;
        }
        return max;
    }
}
```

Next Week: Trees