Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — September 20, 2019
Instructors: Jason Ku, Julian Shun, and Virginia Williams — Problem Set 3

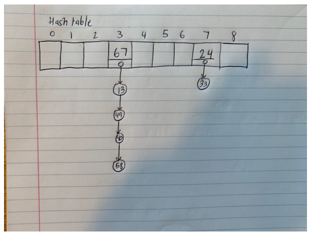
Problem Set 3

All parts are due September 27, 2019 at 6PM.

Name: Diego Escobedo

Collaborators: Noah Lee

Problem 3-1.

(a) 

(b) The Longest chain that we have to traverse to find an element is of length 5, and consosts of $\{67, 13, 49, 40, 58\}$

(c) Using some simple python code, we find that the answer is 13. Here is the code:

```
def find_min_c(arr, c):
    hash_table = [-1]*9
    for elem in arr:
        tr = ((elem*11+4) %c)%9
        if hash_table[tr] == -1:
            hash_table[tr] = elem
        else:
            return False
    return True

i = 1
arr = [67, 13, 49, 24, 40, 33, 58]
while find_min_c(arr, i) == False:
    i += 1
print(i)
```

Problem 3-2.

For our four operations:

Operation 1 is going to be the same as a normal build. The only difference is that now, since we have to have an intrinsic order, we are going to have to hash the index of the thing we're inserting, and where we would usually insert the value, we are going to insert the actual thing. So, we are going to hash the "thing's" index, and not the thing itself.

Operation 2 is pretty simple. We can just feed the index i into our hashing function, which will point us to the exact place in the memory that we need to go to. This is a constant time operation. Once we are there, we can simply find what is there or replace it with the new value in constant time.

Operation 3 is where it starts to get a little trickier. The problem here is that, just from the index, we have no way of telling what the value of the 'thing' stored at that index is. So, we have to visit the thing at index 0 and compare it to our thing we want to insert, until we find the first index where the thing in that index comes after our thing we want to insert. After that, we know where we have to insert our thing, but we still have to move everything else. So, we would revisit every subsequent index, store the thing that was at that index, and move it to the index higher up, so on and so forth. Therefore, this operation will ALWAYS have n steps, meaning that its best and worst case runtime are both O(n).

Operation 4 is the hardest, but if we apply the same principles that we used when amortizing with arrays, we can achieve this O(1) runtime. Similar to what we did before, we can allocate additional slots before and after the numbers that are currently hashed. This allows our operations to be O(1) amortized when we wish to insert and delete at first and last. Further, we also need to keep track of the first and last occupied hash number, because the chances are that our index 0 will not hash to 0. Therefore, if we want to access the first and last element we need to keep track of them somehow. This will only take up constant space.

Problem 3-3.

(a) If we add n to every number in our list of IDs, we will have numbers in the range from 0 to 2n. From here, we can do a radix sort (which we proved in recitation takes O(n) time), which is just about as efficient as a sort can get.

(b) This is going to be another clever application of radix sort. However, since the things to sort are not integers, we need to convert them to integers before. We can do this by using 'base 26' numbers (like hexademical is to base 16 but to base 26). This conversion will only take O(n) time because converting each string only takes constant time. We then know that the longest integer in the set would be 26^{10logn}. By log properties, we know that this largest value is $\leq n^c$ where c is a constant. After that, we can simply perform a radix sort in O(n) time. The dominting term in the addition of all this time complexity is just O(n), so overall this is a O(n) operation.

(c) We are told that the maximum number in this set is going to be n^2. We know that radix sort is guaranteed to take O(n) time if and only if the largest number is less than equal to base n^c if c is a constant. If we use base n in this case, then we can use radix sort in our guaranteed short time because our c would be 2.

(d) The biggest problem here is that we can't radix sort fractions. So, what we need to do is convert all these fractions into integers. To find a single number that will convert all these fractions into integers, we can simply traverse all the numbers and multiply all the denominators together (this takes O(n) because we have to traverse the entire array). We then store this number and multiply every single number in the sequence by this. This way, all the numbers will now be integers and they will maintain their sorting order (correctness!!!). Since the number of fights is capped at n^2, and we proved above that that takes at most O(n) time, we know this will only take O(n) time.

Problem 3-4.

(a) To begin this problem, we're going to need to build a hash table that we are going to use in parallel to the list of side lengths we are given. We are going to use the side lengths as the inputs to our hashing function, and we are going to store boolean True inside the space that corresponds to this hash in the table. This way, we'll have constructed a hash table in O(n) time. Then, the next step is to go through the elements of the list one by one. For each one, we are going to do a O(1) operation of trying to lookup in our hash table. If we are currently at element s_i in our list, we are going to feed (h-s_i) into our hashing function. If there is a match, the location in the hash table will have True stored there. Worst case, we'll have to go through every element in the list (except the last one), which is O(n). Therefore, the whole algorithm runs in O(n). We know that this algorithm will be correct because for every single value in that list of sides, we are looking for its 'complement' which would make h in constant time, so it will happen n times.

(b) For this to work, we are going to need S as a sorted array. Since h = $600n^6$, and we proved above that we can radix sort in O(n) time if we convert to base n and the biggest number is in the order of n^c if c is constant, we know we can radix sort this in O(n) time because c = 6 in this case. Now, we need to come up with some procedure to traverse the array in O(n) time and spit out an answer as to what the smallest difference is. What we need to do is have 4 things stored: a variable diff that keeps track of the smallest difference, a left pointer that starts from arr$[0]$ and a right pointer that starts from arr$[-1]$. The procedure would run as follows:

```
lft = 0
rgt = len(arr) - 1
diff = math.inf
ret = (None, None)
while lft != rgt: #this will terminate when they meet
    if h > arr[lft]+arr[rgt]: #sum isnt bigger than h
        if (h-arr[lft]-arr[rgt]) < diff: #new best diff
           ret = (l, r)
           diff = h-arr[lft]-arr[rgt]
        lft += 1 #lets see if we can make the difference closer
    elif h < arr[lft]+arr[rgt]: #oops, we went over
        rgt -= 1
```

This would guarantee O(n) runtime because we are traversing the array once, and correctness because we are checking every pairing.

Problem 3-5.

(a)

(b)

(c)

(d)

(e)

(f) Submit your implementation to `alg.mit.edu`.