

Solution - 3

1.

Under the hash function $h(x) = x \bmod 10$, all the keys collide.

If I randomly choose my hash function from the universal hash function family we discussed in class I expect 115 and 2545 to collide with probability $1/10$. I randomly chose: $a = 4253$, and $b = 580$.

What really happened was $(4253 * 115 + 580) \bmod 10007 \bmod 10 = 9$ and $(4253 * 2545 + 580) \bmod 10007 \bmod 10 = 8$. There was a 1 in 10 chance they collided, and for this choice of a and b they did not collide.

Using this same hash function on the other keys:

$$(4253 * 995 + 580) \bmod 10007 \bmod 10 = 1$$

$$(4253 * 5025 + 580) \bmod 10007 \bmod 10 = 0$$

I had no collision.

Expected number of collisions $\leq 1000/2000 = \frac{1}{2}$, since there are already a 1000 keys present in the hash table.

2.

Using corollary 11.11 and 11.12 from the text book

a. P [using more $8n$ space]

$$= P \left[E \left[\sum_{i=0}^{m-1} (n_j)^2 > 8n \right] \right] \leq \frac{2n}{8n} = \frac{1}{4}$$

So the probability is $1 - 1/4 = 3/4$

b. P [using more kn space]

$$= P \left[E \left[\sum_{i=0}^{m-1} (n_j)^2 > kn \right] \right] \leq \frac{2n}{kn} = \frac{2}{k}$$

So the probability is $1 - 2/k = (k-2)/k$

3. When we store n keys in a hash table of size $m = n^2$ using a hash function h randomly chosen from a universal class of hash function, the probability is less than $1/2$ that there is any collision. Therefore, probability that there are no collisions is larger than $1/2$. So, $p(n) > 1/2 + (1/2)^2 + (1/2)^3 + \dots + (1/2)^n > 99.9999\%$

$$\Rightarrow 1 - (1/2)^n > 99.9999\%$$

$$\Rightarrow n > 19$$

$$\Rightarrow n = 20$$

So, we try 20 times.

4.

No; For example, $x_1 = 1, x_2 = 6, m = 5, \Pr[h(x_1) = h(x_2)] = 1$

5. Good News ! Since we covered hashing in lecture 3, we now onwards have the incredibly powerful hash-based structures (dictionary, hashset, hashtable etc) available to us to use as subroutines.

Two pieces go together if they weigh w , so $w_1 + w_2 = w$ means the pieces go together. Let us use “weights” to represent the array containing all these pieces.

We will parse through the array once and put each of the items into a hash-structure. We will also calculate the frequency of the pieces if we see similar pieces. This parses “weights” once and so in $O(n)$.

Now we will go through the keys in the HashMap and for each key w_1 find the “opposite” corresponding piece $(w - w_1)$ (using get/search $O(1)$) and delete that element ($O(1)$). w_1 and w_2 are now two valid pieces that can be put back together. In a worst-case scenario, where each piece was unique we will iterate over $n/2$ “ w_1 ” pieces, and find and delete ($O(1)$) their corresponding $n/2$ pieces “ w_2 ”. This makes the algorithm $O(n)$

6. At the core of this question is the sorting algorithm “counting sort” which you can learn more about here: https://en.wikipedia.org/wiki/Counting_sort

SORT_BY_RANK (*array recruits, integer k, integer n*)

```

1) Initialize array output of size n           // the range of output is 1 to n inclusive.
2) Initialize array counter of size k         // the range of counter is 1 to k inclusive.
3) for i = 1 to k                             // initialize the counter array to 0
4)     counter[i] = 0
5)
6) for i = 1 to n                             // count the number recruits in per rank
7)     current_recruit = recruits[i]
8)     counter[current_recruit.rank] += 1
9)
10) cumulative_sum = 0
11) for i = 1 to k:                           // find the cumulative sum of counter
12)     cumulative_sum += counter[i]
13)     counter[i] = cumulative_sum
14)
15) for i = 1 to n
16)     current_recruit = recruits[i]
17)     index = counter[current_recruit.rank] // find the index of current_recruit from counter
18)     counter[current_recruit.rank] -= 1    // decrement counter[rank] so that the next time we
19)                                           // encounter the same rank, it gets index - 1
20)     output[index] = current_recruit      // set the output value
21)
22) return output

```

Assumptions: The input consists of an array of recruit *objects* that needs to be sorted according to their rank, which we can find from a recruit object’s rank attribute, the highest rank “ k ” and “ n ” the size of the input array. For example, if “ x ” is a recruit object, “ $x.rank$ ” would be the rank of “ x ”.

Time complexity analysis: There are 4 loops. 2 of them loop for k times and the other 2 loop for n times. All other code takes constant amount of time. Thus, the time complexity is $O(2n+2k) = O(n+k)$

7.

```
class recruit():
```

```
    def __init__(self, name, colony):
```

```
        self.name = name
```

```
        self.colony = colony
```

```
def sort_colonies(input,c,n):
```

```
    result=[]
```

```
    for i = 1 to n:
```

```
        if(result[input[i].colony] != NULL):
```

```
            result[input[i].colony].count+= 1;
```

```
        else:
```

```
            result[input[i].colony].id = input[i].colony;
```

```
            result[input[i].colony].count= 1;
```

```
    Mergesort(result);
```

```
    return result;
```

Building the result array costs $O(n)$ and mergesort costs $O(c \log c)$, thus the total cost is $O(n + c \log c)$

8.

a. (5 points) Probability that a person on guest list was put in prison is 0. Because the hash function always generate the same hash value for the same input. Therefore, if a person is on the guest list, the value on his position on the hash table must be 1.

b. (7 points) In order for a person who was not on the guest list to get into the party, its hash value might collide with someone who was invited. Therefore, this question is asking us to compute the probability of collision.

Assume the size of hash table is T , and there are k locations set to be 1. Since the hash function we use here follows the simple uniform hashing assumption, the probability of collision is k / T .

c. (8 points) The table size will remain the same. In this case, for every guest i , we will use two hash functions.

$$v1 = h1(i)$$

$$v2 = h2(i)$$

Then we will set both $v1$ and $v2$ to 1. From now on, in order to accept a guest, both locations $v1$ and $v2$ have to be 1.

Now the probability of collision is $(k / T)^2$. Since the $k / T < 1$, so we have reduced the probability of collision. The probability that only people on the guest list can enter is now $1 - (k/T)^2$

FYI, this technique is called bloom filter. You can learn more about it here:
https://en.wikipedia.org/wiki/Bloom_filter