ADS 2020: Week 3 Solutions

Solutions for week 3 of Algorithms and Data Structures. Note that 1.4.5 and 1.4.6 are also covered in the quiz. They are included here as the students may still benefit from discussing why the answers are what they are.

1.4.5, abcd - Green .

a. $\sim N$
b. ~ 1 (because the second term goes toward 0)
c. ~ 1 (because the second and third term goes toward 0)
d. $\sim 2N^3$

1.4.6 - Green .

a. $n + n/2 + n/4... + 1 = 2N - 1 = O(N)$ (geometric sum on page 185)
b. $1 + 2 + 4 + ... + N - N = 2N - N - 1 = N - 1 = O(N)$ (geometric sum on page 185)
c. Inner loop runs N times and outer loop runs $\lg(N)$ times, so $O(N \lg N)$

1.4.10 - Green Everything is done as before until you reach the element you are looking for. At that point, you continue searching for the element in low -> middle. When that search returns notFound, you return middle upwards as you unwind the recursive calls. Notice that doing a linear search after finding the element first time has a worst case running time of $O(N)$ (example: all numbers are identical) and will therefore not satisfy the requirements.

1.4.12 - Green If the current array values are the same, print the value and advance both array indices. Else advance the array index which has the lowest current array value. Break when either of the array indices are out of bounds.

1.4.21 - Green In the constructor of the set, after the call to Arrays.sort(a), go through the a array and count the number of different integers it contains (for instance by counting how many times a[i] is different from a[i-1]). Then make a new array of that size and copy over the contents of the a array while skipping any duplicates. End by assigning a to the new, smaller array. When the StaticSetofInts is now constructed, its internal array has R elements and a standard binary search will run in $\sim lgR$.

1.4.28 - Green Push() just calls enqueue directly. Pop() dequeues and then immediately enqueues all elements in the queue except for the last last element which it dequeues and returns.

1.4.5, efg - Yellow .

e. ~ 1
f. ~ 2
g. There are a couple of things to note. The first is that $\lim_{N\to\infty} N^{100}/2^N = 0$. The second is that $\sim$, given its definition, is undefined for 0 and that 0 can therefore not be the correct answer. Therefore, perhaps somewhat surprisingly, the only function $g(x)$ that satisfies

$$\lim_{N\to\infty} \frac{N^{100}/2^N}{g(x)} = 1$$

is the function itself, $N^{100}/2^N$.

1.4.1 - Yellow

```
Define the number of different triples that can be chosen from N elements as P(N).
Assume that P(N) = N(N - 1)(N - 2) / 6 and consider the base cases.

N = 1       P(N) = 1(1 - 1)(1 - 2) / 6 = 0
N = 2       P(N) = 2(2 - 1)(2 - 2) / 6 = 0
N = 3       P(N) = 3(3 - 1)(3 - 2) / 6 = 1

And we can see that P(N) holds for those.

Now if we assume that:

P(N) = N(N - 1)(N - 2) / 6

and we prove that:

P(N + 1) = (N + 1)N(N - 1) / 6 [*]

then P(N) holds for any arbitrary N.

P(N + 1) = P(N) + (number of new triples that contain the newly inserted N + 1)

There are exactly "N choose 2" new triples that contain N + 1, so:

P(N + 1) = P(N) + N! / 2!(N - 2)!
         = P(N) + N(N - 1) / 2
         = N(N - 1)(N - 2) / 6 + N(N - 1) / 2
         = N(N - 1)(N - 2) / 6 + 3N(N - 1) / 6
         = N(N - 1)(N - 2 + 3) / 6
         = N(N - 1)(N + 1) / 6

And [*] is proven.
```

1.4.24 - Yellow ($\lg N$ solution) Do a binary search. If the egg is not broken at the current floor, F is a on a higher floor. If the egg is broken, we can reuse the solution from 1.4.10 to find the lowest floor level where the egg breaks.

($2 * \lg F$ solution) We now start at the first floor and double the floor number until the egg breaks. This takes at most $\lg F$ times. Then we do a binary search from the last floor the egg didn't break, up to the floor it did break. Which takes $\lg F$ for a combined time of $2 * \lg F$. Notice that the last floor the egg didn't break at is halfway up to the level it broke (as we are doubling the floor number) and we can therefore not search a larger part of the array than F in the binary search.

1.4.25 - Red ($2\sqrt{N}$ solution) Drop the egg at floor $\sqrt{N}$, then at floor $2\sqrt{N}$, $3\sqrt{N}$ etc until the egg breaks. This takes at most $\sqrt{N}$ throws. Because it didn't break at $(i-1)\sqrt{N}$ but did at $(i)\sqrt{N}$, start a linear search upwards at $(i-1)\sqrt{N}$ and it will take at most $\sqrt{N}$ throws until it breaks again for a total cost of $2\sqrt{N}$ throws.

($\sim c\sqrt{F}$ solution) Start by throwing an egg at floor 0, then at floor 1, 3, 6, 10, 15, ... Then:

1. At throw 0, $T_0 = 0$
2. At throw $i+1$, $T_{i+1} = T_i + i = \sum_{j=0}^{i} j \sim= \frac{1}{2}i^2$, where i is the number throws we have currently made as well as the difference between T_i and T_{i+1}

We must therefore show that:

- $i < \frac{c}{2} * \sqrt{F}$

as we first make $i+1$ throws to find floor number $i+1$ and then do a linear search in the i floors between floor T_i and T_{i+1}.

We know that F is between floor i and $i+1$ if the egg did not break at i but did break at $i+1$. In that case,

- $T_i < F$
- $T_{i+1} > F$.

From [2] we know that $T_i \sim= \frac{1}{2}i^2 - i$. Which is $\sim \frac{1}{2}i^2$. Therefore:

$$\sim \frac{1}{2}i^2 < F \tag{1}$$

$$\sim i^2 < 2F \tag{2}$$

$$\sim i < \sqrt{2}\sqrt{F} \tag{3}$$

And we have shown that our algorithm finds the solution in $\sim c\sqrt{F}$ with $c = 2$.

1.4.18 - Red First check if middle is a local minimum. If not, check if the element to the left is smaller. If it is, there must be a local minimum in that part of the array (remember the edge case). If the item to the left is not smaller, and if we are not in a local minimum, then there must be a local minimum in the right side of the array. We continue with a binary search in the correct side of the array.

1.4.29 - Red Keep two stacks, a headstack and a tailstack.

- Enqueue to tailstack.push(). This is a constant time operation.
- Push() to headstack. This is a constant time operation.
- Pop() from headstack. If headstack is empty, headstack.push(tailstack.pop()) until the tailstack is empty. If the headstack is not empty, this is a constant time operation. If it is empty, the operation takes N.

The easiest way to see that this is amortised constant time is to consider how many operations may at most be performed on each element pushed or enqueued onto the steque. For an element pushed, popping it takes 1 operation. An element enqueued will first be popped from the tailstack and then pushed to the headstack, before being popped from the headstack. All constant time operations which are all happening at most one time while the element is in the data structure. So although a single pop() may take O(N), the total operations performed for N elements is O(N) and therefore amortised constant time.

1.4.30 - Red

- PushLeft calls push on the stack.
- PopLeft calls pop on the stack unless it is empty. If the stack is empty, it transfers over half of the elements from the stequeue by first calling steque.enqueue(steque.pop()) for half the elements in the stequeue, and then stack.push(steque.pop()) for the remaining elements. Transferring half the elements of the steque each time the stack is empty is what makes the algorithm run in amortized O(1).
- PushRight calls push on the steque.
- PopRight calls pop on the steque unless it is empty. If it is empty, it transfers ALL the elements from the stack by calling steque.push(stack.pop()).

Analysis of running time: Start by considering the case where we call pushRight N times and then removes the N elements by calls to popLeft(). The first call to popLeft transfers half the elements to the stack for a cost of O(N). At this point, we can pop at least half the elements in O(1) until we need to balance the stack and stequeue again. Assume we do N/2 popLeft calls, and then one more popLeft. Now we do N/2 operations to balance the data structure, then N/4, N/8 etc. This is the geometric sum on page 185 of the book, which sums to O(N). So pushRight and popLeft on N elements has a total cost of O(N), resulting in amortised constant time.

Let's now consider the opposite, calling pushLeft N times and popRight N times. On the first call to popRight, we transfer ALL elements from the stack to the stequeue for a cost of O(N) (Note: Call this situation A) and can now pop them in constant time. We are again using O(N) operations for N elements, resulting in amortised constant time.

Notice that Situation A in the second example is the same situation as after calling pushRight N times, which we showed in the first example is amortised constant time in the case of a call to popLeft. The additional O(N) operations to get here does not change the amortised constant time.

If we interleave pushLeft and pushRight, our popLeft and popRight are cheaper than the worst cases above. So each operation takes amortised constant time.

1.4.31 - Red Keep three stacks, a leftStack, tempStack and rightStack.

- pushLeft to leftStack.
- pushRight to rightStack.
- popLeft from leftStack until empty, then:
  1. Move half of the items from rightStack to tempStack
  2. Move the remaining items from rightStack to leftStack
  3. Return all items from tempStack to rightStack
- popRight from rightStack until empty then:
  1. Move half of the items from leftStack to tempStack
  2. Move the remaining items from leftStack to rightStack
  3. Return all items from tempStack to leftStack

Note on correctness: It's crucial that we use the tempStack for the items that should remain on the non-empty stack to ensure they remain in the correct order after the re-balancing operation completes.

Analysis of running time: Similar to 1.4.30. Moving half the elements each time one of the stacks are empty ensures that we can pop at least N/2 elements in constant time before having to re-balance again. The maximum number of re-balancing operations we can do for N elements is therefore O(N), giving us a constant amortised number of stack operations per call.