ADS 2021: Week 9 Solutions

Solutions for week 9 of Algorithms and Data Structures.

2.3.2 - Green

								a[]						
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11
			E	A	S	Y	Q	U	E	S	T	I	O	N
0	2	11	E	A	E	Y	Q	U	S	S	T	I	O	N
0	1	1	A	E	E	Y	Q	U	S	S	T	I	O	N
0		0	A	E	E	Y	Q	U	S	S	T	I	O	N
3	11	11	A	E	E	N	Q	U	S	S	T	I	O	Y
11		11	A	E	E	N	Q	U	S	S	T	I	O	Y
3	4	10	A	E	E	I	N	U	S	S	T	Q	O	Y
3		3	A	E	E	I	N	U	S	S	T	Q	O	Y
5	10	10	A	E	E	I	N	O	S	S	T	Q	U	Y
5	5	9	A	E	E	I	N	O	S	S	T	Q	U	Y
6	7	9	A	E	E	I	N	O	Q	S	T	S	U	Y
6		6	A	E	E	I	N	O	Q	S	T	S	U	Y
8	9	9	A	E	E	I	N	O	Q	S	S	T	U	Y
			A	E	E	I	N	O	Q	S	S	T	U	Y

2.3.3 - Green

For the implementation of quicksort described in the book, where v is always the first element in the (sub)array, the maximum number of exchanges for the largest element is $\frac{N}{2}$

2.3.8 - Green

Each partition will make N compares and divide the array in half and repeat, So the total number of compares will be about N lg(N).

5.1.2 - Green

input	d=1	d=0	output
no	pa	ai	ai
is	pe	al	al
th	of	co	co
ti	th	fo	fo
fo	th	go	go
al	th	is	is
go	ti	no	no
pe	ai	of	of
to	al	pa	pa
co	no	pe	pe
to	fo	th	th
th	go	th	th
ai	to	th	th
of	co	ti	ti
th	to	to	to
pa	is	to	to

5.1.3 - Green

input	d=0	d=1	output
no	al	ai	ai
is	ai	al	al
th	co	co	co
ti	fo	fo	fo
fo	go	go	go
al	is	is	is
go	no	no	no
pe	of	of	of
to	pe	pa	pa
co	pa	pe	pe
to	th	th	th
th	ti	th	th
ai	to	th	th
of	to	ti	ti
th	th	to	to
pa	th	to	to

2.3.5 - Green

We can use the same principle as the partitioning algorithm in the book. We choose the first element of the array as our pivot element, and then we scan from left to right until we find an array entry that is not equal to our pivot element. Then we scan from right to left until we find an element that is equal to our pivot element and then we exchange the two. Then we continue to do this until the searches "meet" (the indices are equal) and we will have sorted the array.

2.3.4 - Yellow

Any sequence of 10 elements that is already sorted in ascending or descending order is fine here.

2.3.13 - Yellow

- Best case: perfectly partitioned in half each level, meaning the recursion depth will be $\lg N$ in the input size.
- Worst case: partitioned on minimal or maximal element at each recursive step, giving a recursion depth of N.
- Average case: Notice that the immediately intuitive answer of $1.39 \lg N$ can only be considered a lower bound as this is the amount of work done across all recursion levels across all branches of the recursion tree, and not the depth of the deepest recursion for a given input. It is however still the case that the average recursion depth would be some multiple of $\lg N$, but finding an exact number would not be expected.

Old exam set 120531: 3(d-j) - Yellow

(d) - [A]

(e) - [G]

(f) - [B]

(g) - [D]

(h) - [E]

(i) - [C]

(j) - [F]

5.1.17 - Yellow Assuming that R is regarded as constant (since otherwise key counting is irrelevant), we can compute the counts of each key in array just like the version in the book. Then we make a copy, *startIndex*, of the count array that we can use to determine if an element is in its "proper place". Then where the version in the book moves the element to the correct entry in the *aux* array, we instead start at the first array entry, exchange the element to its rightful place within the array (and increment the relevant entry in count[]), and then consider the element that we just did the exchange with. We continue to this until the element is in the correct array entry (i.e. between startIndex[i] and count[i]) at which point we move to the next array entry and repeat until we reach the end of the array.

This version of key indexed counting is not stable however. Consider the input:

(1, "Peter"), (1, "Sumail"), (0, "Artour")

Running this version of key indexed counting results in:

(0, "Artour"), (1, "Sumail"), (1, "Peter")

Which is sorted according to the keys but not kept in the same relative order for items with equal keys.

2.3.17 - Red See code below

```
public class Quick {
    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a); // Eliminate dependence on input.\
        // Find the index of the largest element in a
        // and use it as sentinel by placing it at end.
        int indexOfMax = findMaxIndex(a); // <- assume we have this.
        exch(a, indexOfMax, a.length-1);
        sort(a, 0, a.length - 1);
    }
    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1); // Sort left part a[lo .. j-1].
        sort(a, j+1, hi); // Sort right part a[j+1 .. hi].
    }
    private static int partition(Comparable[] a, int lo, int hi) {
        // Partition into a[lo..i-1], a[i], a[i+1..hi].
        int i = lo, j = hi+1; // left and right scan indices
        Comparable v = a[lo]; // partitioning item
        while (true)
        { // Scan right, scan left, check for scan complete, and exchange.
            // Now we can remove the two bound checks from
            // the while loops below.
            while (less(a[++i], v)) break;
            while (less(v, a[--j])) break;
            if (i >= j) break;
            exch(a, i, j);
        }
        exch(a, lo, j); // Put v = a[j] into position
        return j; // with a[lo..j-1] <= a[j] <= a[j+1..hi].
    }
}
```

2.3.15 - Red To solve this problem, we can take an arbitrary bolt and compare it with each nut. If the nut is smaller than the bolt, we put it to the left, if it is larger than the bolt, we put to the right and if it fits we put it right next to the bolt (but still compare all nuts to the bolt). Once we are done, we take a new random bolt, and first compare it to the nut that fit the bolt, to determine if it is smaller or larger than the first bolt. If it is smaller, then we repeat the process for the partition of nuts that were smaller than the first bolt and otherwise we do it on the partition that was larger. Using this approach we can do a binary search on the partitions to find the relevant partition for each new bolt that we pick up, which is significantly more efficient than comparing each unused bolt to each unused nut.

Note that we must first shake the bag in such a way that smaller bolts and nuts do not float to the top of the bag - achieving a randomly distributed pile of nuts and bolts before we start the algorithm.