Tutorial 3

Problem 1: Quicksort review

a) Suppose that the pivot choice is the median of the first, middle and last keys, can you find a bad input for QuickSort?

All the keys in the array are the same.

> As long as we have a fixed pivot, we can get a algo with a O(n^2) worst case run time. We can always find a bad input.

---

b) Are any of the partitioning algorithms we have seen for QuickSort stable? Can you design a stable partitioning algorithm? Would it be efficient?

No, none are. However, it is possible. You can have two arrays, one to store the partitioned version and the original array. Pick a pivot, then iterate through the original array: If the element is smaller, add it to the partitioned array. Finally, all you are left with are the pivot and the elements which are greater. Add that on accordingly.

However, the space complexity is much worse as everytime you partition, you create another array. In the worse case scenario, the space complexity is O(n^2).

Time complexity is the same.

We can also have 2 arrays, one to keep track of larger elements and the other to keep track of smaller elements.

(c) Consider a QuickSort implementation that uses the 3-way partitioning scheme (i.e. elements equal to the pivot are partitioned into their own segment).

i) If an input array of size n contains all identical keys, what is the asymptotic bound for QuickSort?

O(n) time.

ii) If an input array of size n contains k < n distinct keys, what is the asymptotic bound for QuickSort?
For example, with n = 6, k = 3, sort the array [a, b, a, c, b, c]

O(n logk)

Maximum of k pivots to choose from.

Problem 2: (A few puzzles involving duplicates or array processing)

(a) Given an array A, decide if there are any duplicated elements in the array.

Use a HashSet

```
import java.util.HashSet;
```

```
public class CheckDuplicate {
  public static boolean hasDuplicate(int[] array) {
    HashSet<Integer> set = new HashSet<>();
    for (int i = 0; i < array.length; i++) {
      if (!set.add(array[i])) {
        return true;
      }
    }
    return false;
  }
}
```

Arrays solution

```
import java.util.Arrays;

public class CheckDuplicateNoHash {
    public static boolean hasDuplicates(int[] A) {
        Arrays.sort(A); // sort the array
        for (int i = 1; i < A.length; i++) {
            if (A[i] == A[i - 1]) { // check for duplicates
                return true;
            }
        }
        return false;
    }

}
```

(b) Given an array A, output another array B with all the duplicates removed. Note the order of the elements in B does not need to follow the same order in A. That means if array A is {3, 2, 1, 3, 2, 1}, then your algorithm can output {1, 2, 3}.

```
import java.util.Arrays;
import java.util.HashSet;

public class RemoveDuplicates {
    public static int[] removeDuplicates(int[] A) {
        HashSet<Integer> set = new HashSet<>();
        for (int x : A) {
            set.add(x);
        }
        int[] B = new int[set.size()];
        int i = 0;
        for (int x : set) {
            B[i++] = x;
        }
        return B;
    }

    public static void main(String[] args) {
        System.out.println(Arrays.toString(removeDuplicates(new int[]{1, 2, 5, 3, 4, 6, 5})));
    }
}
```

This algorithm has a time complexity of O(n + m) where n is the number of elements in the input array A. The space complexity is also O(n + m).

(c) Given arrays A and B, output a new array C containing all the distinct items in both A and B. You are given that array A and array B already have their duplicates removed.

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

public class ArrayMerger {
    public static int[] mergeArrays(int[] A, int[] B) {
        Set<Integer> set = new HashSet<>();
        for (int i : A) set.add(i);
        for (int i : B) set.add(i);
        int[] C = new int[set.size()];
        int i = 0;
        for (int n : set) C[i++] = n;
        return C;
    }

    public static void main(String[] args) {
        System.out.println(Arrays.toString(mergeArrays(new int[]{1, 2, 3, 4, 5}, new int[]{2, 3, 5, 6, 7})));
    }
}
```

Time complexity is O(n)

Space complexity is also O(n).

If I cannot use a HashSet:

1. Sort both arrays A and B in increasing order.
2. Initialize two pointers, i for array A and j for array B.
3. Initialize an empty array C to store the result.
4. While i < A.length and j < B.length, compare A[i] and B[j].
5. If A[i] is less than B[j], increment i.
6. If B[j] is less than A[i], increment j.
7. If A[i] is equal to B[j], add A[i] to C and increment both i and j.
8. Repeat steps 4 to 7 until one of the arrays has been fully traversed.
9. Add the remaining elements in the other array to C.
10. Return C.

Time Complexity: O(nlogn)

Space Complexity: O(n) as we need to store the result in a new array C of max size n

```
import java.util.Arrays;

class ArrayMergerNoHash {
    public static int[] mergeArraysNoHash(int[] A, int[] B) {
        Arrays.sort(A);
        Arrays.sort(B);
        int i = 0, j = 0, k = 0;
```

```
        int[] C = new int[A.length + B.length];
        while (i < A.length && j < B.length) {
            if (A[i] < B[j]) {
                C[k++] = A[i++];
            } else if (B[j] < A[i]) {
                C[k++] = B[j++];
            } else {
                C[k++] = A[i++];
                j++;
            }
        }
        while (i < A.length) {
            C[k++] = A[i++];
        }
        while (j < B.length) {
            C[k++] = B[j++];
        }
        return Arrays.copyOf(C, k);
    }

    public static void main(String[] args) {
        System.out.println(Arrays.toString(mergeArraysNoHash(new int[]{1, 2, 3, 4, 5}, new int[]{2, 3, 5, 6, 7})));
    }
}
```

Whether this solution is more efficient or not than the solution with the HashSet depends on the specific use case and the size of the arrays. In general, sorting the arrays and using two pointers is faster for smaller arrays, but using a HashSet is faster for larger arrays where the cost of inserting elements into a HashSet becomes insignificant compared to the cost of sorting and using two pointers. So, it is best to choose a solution based on the size of the arrays and the specific requirements of the use case.

(d) Given array A and a target value, output two elements x and y in A where (x+y) equals the target value.

```
import java.util.Arrays;

public class MakeTarget {
    public static int[] twoSum(int[] A, int target) {
        Arrays.sort(A);
        System.out.println(Arrays.toString(A));
        int start = 0, end = A.length - 1;
        int[] result = new int[2];

        while (start < end) {
            int sum = A[start] + A[end];
            if (sum == target) {
                result[0] = A[start];
                result[1] = A[end];
                return result;
            } else if (sum < target) {
                start++;
            } else {
                end--;
            }
        }

        return result;
    }

    public static void main(String[] args) {
        int[] A = {8, 1, 2, 3, 4, 5, 6, 7};
```

```
        int target = 9;

        System.out.println(Arrays.toString(twoSum(A, target)));
    }
}
```

O(logn) time. O(1) space.

Problem 3: Child Jumble

O(n^2) brute force lol

Take one kid then let him try on all shoes: partition into the shoes that are too small, too big.

Recurse.

Kids in the “too big group” —> those who find the pivot’s shoe to small

Kids in the “too small grp” —> opposite.

O(nlogn)

Problem 4: More Pivots!

(a) Suppose that you have a magic black box function which chooses k perfect pivots that separate the elements evenly (e.g. it picks the quartile elements when there are 3 pivots). How would a partitioning algorithm work using these pivots?

A quicksort partitioning algorithm using magic black box pivots would work as follows:

1. The algorithm would then partition the array into k sub-arrays by comparing each element in the array to the k pivots.
2. Each sub-array would be further partitioned using the magic black box function to choose new perfect pivots that separate the elements evenly.
3. The process of partitioning would be repeated recursively until each sub-array only contains one or a few elements.
4. The final sorted array would be obtained by concatenating all of the sub-arrays in order.

Can also use binary search (O(nlogk)) to find the buckets.

(b) What is the asymptotic running time of your partitioning algorithm? Give your answer in terms of the number of elements, n and the number of pivots, k.

O(nlogk)

---

c)

T(n) = (k+1)T(n/k) +O(nlogk)(As k pivots are perfectly chosen. We can also use kT(n/k) instead. k+1 is the number of partitions. each time we do nlogk as we are doing binary search)

---

d)

Draw the tree

O(nlogk)

2 and 3 pivot quicksort is faster not because of algorithm analysis, but it utilises the cache better.

Java uses 2 pivot quicksort.

3 is faster than 2 pivots.

Problem 5: Integer Sort

(a) Given an array consisting of only 0's and 1's, what is the most efficient way to sort it? (Hint: Consider modifying a sorting algorithm that you have already learnt to achieve a running time of O(n) regardless of the order of the elements in the input array.)
Can you do this in-place? If it is in-place, is it also stable? (You should think of the array as containing key/value pairs, where the keys are 0's and 1's, but the values are arbitrary.)

1. Initialize two pointers, one at the start of the array (left pointer) and one at the end of the array (right pointer).
2. Keep incrementing the left pointer until a 1 is found.
3. Keep decrementing the right pointer until a 0 is found.
4. Swap the values of the two pointers.
5. Repeat steps 2 to 4 until the left pointer is greater than the right pointer.

This algorithm has a time complexity of O(n), as each element in the array is visited only once. It also has a space complexity of O(1), as no additional data structures are used. However, it is not stable.

Stable solution:

1. Create two separate arrays, one for each key (0 and 1)
2. Traverse the input array and count the number of occurrences of each key
3. Update the count arrays so that each element stores the cumulative count of the elements before it
4. Traverse the input array again and place each element in its correct position in the output array, based on its count in the count arrays

This algorithm has a time complexity of O(n) and a space complexity of O(n)

(b) Consider an array consisting of integers between 0 and M, where M is a small integer (For example, imagine an array containing key/value pairs, with all keys in the range {0, 1, 2, 3, 4}). What is the most efficient way to sort it? This time, you do not have to do it in-place; you can use extra space to record information about the input array and you can use an additional array to store the output.

Use Counting sort

Go through array and count the frequency (M comes from the frequency array)

Then compute the index in the output array.

Time complexity: O(n + M)

Might be bad in some cases when largest element is like 2^32

If M is small enough then you can get like O(n)

O(M) space complexity (if we don't count the output array)

---

c)

O(n) time to sort each bit level

Number of levels: 64

64n

64 needs to be smaller than logn if we want it to be faster than quicksort

So it will only beat quicksort if n >2^64

---

d)

Do more work per time but less recursive loops

2^k - 1

Refer to slides

Using 8 bit chunks, but it also depends on how many chunks you use. As you increase the chunk size, the number of numbers in you frequency table gets larger.

Also there is Radix Sort which sorts bit by bit. It used counting sort for every iteration. It is also stable.
