Sample Case Study & Solution

1. **Asymptotic Notation**, Lecture - 2
    a. Suppose you have an array of 1234 records in which only a few are out of order and they are not very far from their correct positions. Which sorting algorithm (among merge, insertion, heap, quick, bubble, counting, and radix) would you see to put the whole array in order? Justify your choice.

    Solution: Insertion sort. Suppose k elements are out of order and, among them, the ith (i = 1, 2,...., k) element is d_i away from its correct position. The complexity, in terms of the number of exchanges, is

    $$\sum_{i=1}^{k} d_i$$

    Note that $k \ll n$ and $d_i \ll n$.
2. **Recurrence Relations**, Lecture - 3
    a. Consider the following puzzle. There is a row of n chairs and two types of people: M for mathematicians and P for poets. You want to assign one person to each seat but you can never seat two mathematicians together or they will start talking about mathematics and everyone else in the room will get bored. For example, if n = 3, the following are some valid seatings: PPP, MPM, and PPM. However, the following is an invalid seating: MMP. In this problem, your goal is as follows. Let f(n) be the number of valid seatings when there are n chairs in a row. Write and solve a recurrence relation for f(n). Please show your work.

    Solution: The trick to solving this problem is to realize that you can get a valid seating of n people in two ways. First, you can take a valid seating of n−1 players and then put a P in the n-th seat. This ensures there won't be two M's together. Second, you can take a valid seating of n−2 players, put a P in the (n-1)-st seat and then put an M in the n-th seat. This exhausts all the ways of getting valid seating. Translating this into a recurrence relation gives us that f(n) = f(n − 1) + f(n − 2). When we solve this recurrence relation (with annihilators), we get that f(n) is simply the n-th Fibonacci number.

3. **Divide & Conquer**, Lecture - 4
    a. An array of n elements contains all but one of the integers from 1 to n + 1.
        i. Give the best algorithm you can for determining which number is missing if the array is sorted, and analyze its asymptotic worst-case running time.

           Solution: If the array is sorted, we can use binary search. We are looking for the smallest index i for which A[i] = i+ 1; this will be our missing number. If A[n/2] = n/2 + 1, i is less than or equal to n/2, and we can recurse on the first half of A; otherwise it is greater than n/2, and we can recurse on the second half of A. In either case, we get the recurrence T(n) = T(n/2)+Θ(1) which gives a worst-case running time of Θ(log n).
        ii. Give the best algorithm you can for determining which number is missing if the array is not sorted, and analyze its asymptotic worst-case running time.

           Solution: If the array is not sorted, then in the worst case we will have to look at every location in the array (otherwise what we think is the missing element could be in the location we didn't look in). So the best we can hope to do is get an O(n) algorithm. One such algorithm creates an auxiliary array B with indices 1 to n + 1, initializes all locations to 0, scans through A setting B[A[i]] = 1 for each i from 1 to n, and finally scans through B looking for a zero. Each of these three steps takes O(n) time, so we have an O(n) algorithm for this case.

4. **Merge Sort**, Lecture - 5
   a. Consider the following variant of MergeSort: instead of splitting the list into two halves, we split it into three thirds. Then we recursively sort each third and merge them. This is called three-way MergeSort.
      i. What is the total number of key comparisons performed in the worst case, while merging three sorted lists, each of length n/3, to one sorted list? Also, express your answer using O(•) notation.

         Solution: n/3+n/3 - 1 = 2n/3 - 1 for Merge(B,C,E);

         2n/3 + n/3 - 1 = n - 1 for Merge(E,D,X);

         Total: 5n/3 - 2 $\in$ O(n)
      ii. Let T(n) denote the worst-case running time of three-way MergeSort on an array of size n. Write a recurrence relation for T(n).

         Solution: T(n)=3T(n/3)+O(n)
      iii. Solve the recurrence relation in part (c). Express your answer using O(•) notation.

         Solution: By Master theorem, T(n)=O(n logn)
      iv. Is the three-way MergeSort asymptotically faster than insertion sort?

         Solution: Yes

v. Is the three-way MergeSort asymptotically faster than ordinary MergeSort?

   Solution: No

5. **Quick Sort**, Lecture - 6
   a. You are compiling a list of students (ID, weight) in Singapore, for your CCA. However, due to a budget cut, you are facing a problem with the amount of memory available for your computer. After loading all students in memory, the extra memory available can only hold up to 20% of the total students you have! Which sorting method should be used to sort all students based on weight (no fixed precision)?

      Solution: Quick Sort. Due to memory constraints, you will need an in-place sorting algorithm. Hence, a sorting algorithm that is both in-place and works for floating-point is Quick Sort. Do note that: The system requires some extra space on the call stack, due to the recursive implementation of Quick Sort (and similarly for Merge Sort), although we say that Quick Sort is in-place.
6. **Heap Sort**, Lecture - 7
   a. Suppose that instead of using Build-Heap to build a max-heap in place, the Insert operation is used n times. Starting with an empty heap, for each element, use Insert to insert it into the heap. After each insertion, the heap still has the max-heap property, so after n Insert operations, it is a max-heap on the n elements.
      i. Argue that this heap construction runs in O(n log n) time.

         Solution: Insert takes O(log n) time per operation, and gets called O(n) times.

ii. Argue that in the worst case, this heap construction runs in Ω(n log n) time.

Solution: If you insert the elements in sorted order (starting with 1), then each insert puts the element at a leaf of the heap, before bubbling it up all the way to the root. This takes Θ(log k) swaps, where k is the number of elements already in the heap.

$$\sum_{k=1}^{n} log\ k = \Theta(n\ log\ n)$$

7. **Linear Time Sorting**, Lecture - 8

a. After your success in creating the list for your CCA, you are hired as an intern in NUS to manage a student database. There are student records, already sorted by name. However, we want a list of students first ordered by age. For all students of the same age, we want them to be ordered by name. In other words, we need to preserve the ordering by name as we sort the data by age.

Solution: Radix Sort. The requirements call for a stable sorting algorithm so that the ordering by name is not lost. Since memory is not an issue, Radix Sort can be used. Radix Sort has a lower time complexity than comparison based sorts here, O(dn) where d = 2, vs O(n log n) for Merge Sort.