Lecture 9: Heapsort and Quicksort

I showed how to perform Heapsort using a heap. This part is well explained in the textbook. So I will not repeat it here. Make sure you know how the algorithm runs and how to analyze the running time.

Now Quicksort is yet another sorting algorithm. The way of divide is different from Mergesort: for a given array A, it chooses a pivot p and then creates two sets A− (with elements smaller than the pivot) and A+ (with elements of A larger than the pivot). Then it ensures A− is before p, and A+ is after p in A. This can be done in place (i.e. with no additional memory), which we did not discuss in class. The running time depends on whether the partition is balanced (called lucky case), or unbalanced (unlucky case). In the case where we have two equal size A− and A+, the running time T(n) = 2T((n−1)/2) + Θ(n) ≤ 2T(n/2) + Θ(n).

In this case, T(n) = O(nlogn). In the unlucky case, T(n) = T(n−1) + Θ(n), which leads to T(n) = O(n²).

This can be shown by direct substitution.

A final point: if we divide with A− with n/10 elements and A+ with 9n/10 elements, it turns out this is also a lucky case. To see this, we consider the recursion tree. Each level takes O(n) time in divide and combine. Let the number of levels be k. We trace from the root to the leaf which going through k edges. Each time, the size of subproblem reduces by a factor of 9/10. Since at leaf the size of subproblem is 1, we have n(9/10)^k = 1, i.e. k = log_{10/9}n. So the total running time in this case is O(nlogn).

Lecture 10: Quicksort and Counting sort

Suppose we alternative between lucky and unlucky, we have: L(n) ≤ 2U(n/2) + O(n), U(n/2) ≤ L(n/2−1) + O(n). So L(n) ≤ 2L(n/2−1) + 2O(n/2) + O(n) <= 2L(n/2) + O(n). This leads to O(nlogn) running time. This suggest it is likely that the expected running time of Quicksort is about O(nlogn).

We now do a more rigorous analysis. We randomly choose pivot. We say we have (k,n-k-1) split if the size of A− is k. Note each split is same chance: 1/n. For a (k,n-k) split, we have T(n) = T(k) + T(n−k−1) + O(n).

Thus, E(T(n)) = \sum_{k=0}^n \frac{1}{n} (T(k) + T(n−k−1) + O(n)). So E(T(n)) = \sum_{k=0}^n \frac{1}{n} \left( E(T(k)) + E(T(n−k−1))\right) + O(n) = \frac{2}{n} \sum_{k=0}^n E(T(k)) + O(n). Here we combine E(T(0)) into O(n) term. We claim: T(n) ≤ anlogn for some a, which we show by induction. The base case of n = 2 can be satisfied if a is large enough. We now assume the claim holds for all integers less than n. We need the following mathematical fact: \sum_{k=1}^n klogk ≤ \frac{1}{2}n^2logn − \frac{1}{8}n^2.

From induction hypothesis, E(T(n)) ≤ \frac{2}{n} \sum_{k=1}^{n-1} aklogk + cn ≤ \frac{2}{n} \left( \sum_{k=1}^n k = \frac{1}{2}n^2 \right) + cn, which is ≤ 2/n(1/2n^2logn − 1/8n^2) = anlogn − a/4n + cn. This holds when a is large enough.

Now counting sort. This is explained well in the textbook. One thing to note is that we do need to use the definition of C[i] being equal to the number of array elements smaller or equal to i. This is because it is often not enough to output a list of integers (which can be easily done from the first version of definition of C). For example, a database record contains not just its ID, but also other information like name, address, and so on. So we need to be able to place the original record into the result.

The other important thing to note is how to compute C[i] efficiently. Naively C[i] is equal to \sum_{k=0}^i C_0[k], where C_0[i] is equal to the number of elements of A equal to i. Directly computing like this will not be efficient. Remember: we need to compute each C[i] in constant time. But a simple trick is: C[i] = C[i−1] + C_0[i]. So we will compute C[i] from 0 to k sequentially using C[i] = C[i−1] + C_0[i] (since C[i−1] has already been computed at the time we need to compute C[i]). This concept is crucial: we will use something similar extensively in the later part of this course.