Lecture 9: Heapsort

In this lecture, I first re-visit the Quicksort analysis. We assume the given input is some permutation of the numbers from 1 to n. One key quantity to analyze whether two values i and j are compared or not. The key fact is that this is solely determined by which value within \([i, j]\) is first chosen as pivot. This is because all previous pivots will all make i and j to be in the same subproblem. Thus, exactly two out of \(j - i + 1\) choices of pivot will compare i with j.

We now discuss the HeapSort algorithm. Since most of the students already know what is heap, we will focus on analysis part. See Chapter 6 if you forget what is heap, covered in data structure course. There are several key facts about heap. The most important one is, the height \(Ht\) of the heap is \(\Theta(lgn)\). To see this, we note that at level \(i\), there are no more than \(2^i\) nodes. Here, we say the root is at level 0 (where the root has height \(Ht\)). Also note the heap forms a complete binary tree. Thus, we have the number of nodes \(n \leq \sum_{i=0}^{Ht} 2^i = 2^{Ht+1} - 1\). Also, if we omit the last term in the above summation, we have: \(n \geq 1 + \sum_{i=0}^{Ht-1} 2^i = 2^{Ht} - 1\). Thus, \(lgn - 1 \leq Ht \leq lgn\).

Based on this fact, the algorithm Heapify(A,i) will run \(O(lgn)\) time since it only processes up to \(lgn\) nodes when pushing the small value downwards, each time takes constant time. We conclude with the algorithm for building the heap from an unordered array A by repetitively calling Heapify(A,i).

The next subject is how to build a heap and how to sort an array using a heap. Again, these subjects are well explained by our textbook. Please review them. An important application of the heap is its use in priority queue, a dynamic data structure. In class, I showed how a heap can be used to implement: (i) get the maximum priority in \(O(1)\) time; (ii) extract the element with the maximum priority in \(O(logn)\) time; (iii) increase the priority of an element in \(O(logn)\) time; and (iv) insert a new element with certain priority. Note that increasing priority of an element takes \(O(logn)\) time because we only need to trace upwards in the tree and swap priority values of the current node with its parent if these two values violate the heap property. Also, inserting a new element with some priority can be done through calling increasing priority routine.

Lecture 10: Lower bound, Counting sort and Selection

Now, lower bound for sorting. We discuss the issue of whether \(O(nlogn)\) sorting is optimal. It is, if we assume only comparison is allowed. See Section 8.1 for more details. The basic idea is as follows. Suppose we perform \(d\) comparisons. There are \(2^d\) possible combinations of results of these \(d\) comparisons. Note that we must determine the proper ordering of the \(n\) elements. Since there are totally \(n!\) possible orderings, this implies that \(2^d \geq n!\). Otherwise, some ordering will not be recognized by the sorting algorithm using only \(d\) comparisons, since each comparison result combination can lead to no more than one ordering. Thus, \(d \geq \log(n!)\). Note that \(n! \geq (n/2)^{n/2}\) (by taking the second half of the elements in \(n!\) and then approximate with \(n/2\) for each remaining element), we know, \(d \geq \Omega(nlogn)\). Thus, the running time will be \(\Omega(d) = \Omega(nlogn)\).

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 present 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.

Our next subject is the selection problem. A frequently used statistic is the median, which is sometimes better than average (mean). One example is that there are 1000 numbers, where 999 of them are 1 but one is 10000000. Here, median is more representative than mean. Now we consider the more general problem: finding the k-th smallest element among n elements. Here is the problem formulation: given a list of n elements, and an integer k. Our goal is finding the k-th element in the list. A naive solution is to first sort then output the k-th
item. This can be done in $O(n \log n)$ time. But it seems to waste of a lot of time: we only want to find the k-th item and we do not need to know the full order. Here, we can apply a randomized divide and conquer method. Just like Quicksort, we partition the input list by picking a pivot $v$. For this pivot $v$, we divide the given list into three parts, those smaller than (denoted as $S_l$), equal to ($S_m$), or larger than $v$ ($S_r$). Now, we define $\text{Selection}(S, k)$ as the function that will return the k-th element in list $S$. Then, $\text{Selection}(S, k) = \text{Selection}(S_l, k)$ if $k \geq |S_l|$, and return $v$ if $k > |S_l|$ and $k \leq |S_l| + |S_m|$, and $\text{Selection}(S_r, k - |S_l| - |S_m|)$ otherwise. Just like Quicksort, the running time depends on how evenly the two lists are partitioned. The worst case running time is $O(n^2)$. We call lucky case as the situation where the two subproblems are relatively balanced: if the selected $v$ is within [25%, 75%]. In this case, the two subproblems can not be very unbalanced (at most 0.75 of original size). If you pick randomly, 50% to be lucky. The expected number of parititions needed to achieve a lucky partition is 2. Thus, after two rounds, the size is no more than 0.75. Let $T(n)$ be expected run time. We have: $T(n) \leq T(0.75n) + cn$, and the expected running time is $T(n) = O(n)$. 