Lecture 8: Quicksort

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 pace (i.e. with no additional memory), which we briefly discuss in class. Read pages 171-173 for more details. 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 $T(n) = 2T((n-1)/2) + \Theta(n) \leq 2T(n/2) + \Theta(n)$. In this case, $T(n) = O(n\log n)$. In the unlucky case, $T(n) = T(n-1) + \Theta(n)$, which leads to $T(n) = O(n^2)$. This can be shown by direct substitution.

Now, 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 $O(n\log n)$.

Suppose we alternative between “lucky” and “unlucky”, where lucky means we divide the problem into half and unlucky means we only reduce the problem size by one. Then we have: $L(n) \leq 2U(n/2) + O(n)$, $U(n/2) \leq L(n/2 - 1) + O(n)$. So $L(n) \leq 2L(n/2 - 1) + 2O(n/2) + O(n) \leq 2L(n/2) + O(n)$. This leads to $O(n\log n)$ running time. This suggest it is likely that the expected running time of Quicksort is about $O(n\log n)$.

We now do a more rigorous analysis. Instead of always picking say the last element within the region, we now randomly choose the pivot. The expected running time of this randomized algorithm is analyzed in Section 7.4.2. Here are a list of major points:

1. The running time of Quicksort is proportional to the number of comparisons performed during the execution. This is because for each comparison, we spend constant time (to move elements around during partition, and partition is the main work involved in Quicksort).

2. If element $x$ and $y$ are compared, one of $x$ and $y$ must be a pivot sometime.

3. Moreover, if $x$ and $y$ are ever compared, they will be compared again. This is because the pivot will be fixed in positions and will not be involved later in comparisons. This allows us to define indicator variables for the events that two elements $x$ and $y$ are compared or not. And the frequently used linearity of expectation suggests that we now only need to know how to compute the probability of such event.

4. WLOG assume we have a permutation of 1, 2, ..., n. The final point is, elements $i$ and $j$ (where $i < j$) are compared iff $i$ or $j$ are the first to be chosen as pivot among elements: $i, i + 1, ..., j$. Why? First note that if no pivot is chosen among $[i..j]$, elements from $i$ to $j$ will be within the same partition (why?). And if either $x$ or $y$ is chosen as pivot for the first time among $[i..j]$, $x$ and $y$ will be compared (why?). Then if any pivot $i < k < j$ is chosen as pivot, then $i$ and $j$ will be placed in different partition and will not be compared again. This leads to probability of $2/(j - i + 1)$.

Lecture 9: More Soring: Heapsort, Lower bound and Counting sort

We start with 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 $H_t$ of the heap is $\Theta(\log n)$. 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 $H_t$). Also note the heap forms a complete binary tree. Thus, we have the number
of nodes $n \leq \sum_{i=0}^{H_t} 2^i = 2^{H_t+1} - 1$. Also, if we omit the last term in the above summation, we have:

$$n \geq 1 + \sum_{i=0}^{H_t-1} 2^i = 2^{H_t}.$$  Thus, $lgn - 1 \leq H_t \leq lgn$.

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.

Now, lower bound for sorting. We discuss the issue of whether $O(n logn)$ 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, $d \geq \Omega(n logn)$. Thus, the running time will be $\Omega(d) = \Omega(n logn)$. 