Lecture 9: Quicksort and Selection

We continue with the analysis of QuickSort. 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 \( L(n) \leq 2L(n/2 - 1) + 2O(n/2) + O(n) \). 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: \( i, i + 1, \ldots, j \). 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 \( 2/(j-i+1) \).

5. The above observation allows use to define indicator variables \( X_{i,j} \) for the event “elements \( i \) and \( j \) are compared during the QuickSort algorithm”. Then the total number of comparison is \( X = \sum_{i,j} X_{i,j} \). We can then apply what we have learned about indicator variables. After some simplification, we get the expected running time of the randomized QuickSort is \( O(n \log n) \).

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 1000000. 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 then apply what we have learned about indicator variables. After some simplification, we get the expected running time of the randomized QuickSort is \( O(n \log n) \).

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: \( i, i + 1, \ldots, j \). 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 \( 2/(j-i+1) \).

5. The above observation allows use to define indicator variables \( X_{i,j} \) for the event “elements \( i \) and \( j \) are compared during the QuickSort algorithm”. Then the total number of comparison is \( X = \sum_{i,j} X_{i,j} \). We can then apply what we have learned about indicator variables. After some simplification, we get the expected running time of the randomized QuickSort is \( O(n \log n) \).

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 1000000. 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 element 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 \( T(n) = O(n) \).
Lecture 10: Worst-case $O(n)$ Selection and Heapsort

The first topic is the worst-case $O(n)$ time algorithm for the Selection problem (Section 9.3 in the book). The idea is again using divide-conquer scheme. The main difficulty of Selection is finding a good pivot which allows relatively even partition. For this, we first arbitrarily divide the $n$ numbers into groups of 5 elements (and so there are $n/5$ groups). I ignore the remainders for now (do not matter for asymptotic behavior: see the textbook for more precise version). Then we find the median of the $n/5$ groups using brute force (simple: only 5 elements) and thus $O(1)$ time each and total time for this step is $O(n)$. We then recursively find (using exactly the same worst case $O(n)$ algorithm by making the algorithm finding the median) the median of the medians of the $n/5$ groups; let $x$ be the median of medians. We then partition the original $n$ numbers with $x$ as the pivot. The rest of algorithm is just as the randomized version: it issues recursively call (if needed) to one of the subproblems. The key question is why this algorithm runs in $O(n)$ time. This is because $x$ is ensured to be not too unbalanced. Why? Note that at least $0.5 * (n/5) − 2$ groups with median $x$ or smaller and with 5 elements in each group. So those groups have at least 3 elements larger than its median and so is also at least $x$. So we have at least $3 * (0.5 * (n/5) − 2) = 3n/10 − 6$ elements larger than $x$. We can perform the same logic to the other side of the partition. This implies that the subproblem we are solving recursively will have at most $7n/10 + 6$ elements. So we have: $T(n) ≤ T(n/5) + T(7n/10 + 6) + an$ (for some constant $a$) when $n$ is relatively large, and this shows the time is $O(n)$. Why? We can prove by induction. Assume assume $T(n') ≤ cn'$ for all $n' < n$. Then, $T(n) ≤ c * n/5 + c(7n/10 + 6) + a = 0.9cn + 7c + an = cn + ((−c/10 + a)n + 7c) ≤ cn$, since the last inequality holds when $c$ is large enough.

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 $H_t$ of the heap is $Θ(lgn)$. 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 $H_t$). Also note the heap forms a complete binary tree. Thus, we have the number of nodes $n ≤ \sum_{i=0}^{H_t} 2^i = 2^{H_t+1} − 1$. Also, if we omit the last term in the above summation, we have: $n ≥ 1 + \sum_{i=0}^{H_t-1} 2^i = 2^{H_t}$. Thus, $lgn − 1 ≤ H_t ≤ lgn$.

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.