Adapted From Virginia Williams' lecture notes.

COMP 285 (NC A&T, Spr '22) Lecture 10

QuickSort and Non-Comparison Sorts

Today we will complete our analysis of `Quicksort`, which is the last comparison-based sort we will cover in class.

Then, we will jump into learning more about non-comparison-based sorts, such as `CountingSort` and `RadixSort`.

1 QuickSort Analysis

1.1 Worst-Case Analysis

In this section we will derive a bound on the worst-case running time of `Quicksort`. If we consider the worst random choice of pivot at each step, the running time will be $\Theta(n^2)$. This flavor of worst-case analysis (which gets an upper bound on the running time over all possible possible choices of pivots) is no different from the worst case analysis of the algorithm which picks an arbitrary pivot at every step. We are thus interested in what the running time of `Quicksort` is **on average over all possible choices of the pivots**. We should emphasize an important point: We still consider the running time for a **worst-case input**, and average only **over the random choices of the algorithm** (which is different from averaging over all possible inputs). Put differently, our analysis will guarantee that **for any input, the expected running time will be small**. We formalize the idea of averaging over the random choices of the algorithm by considering the running time of the algorithm on an input I as a random variable and bounding the expectation of that random variable.

Proposition. *For every input array of size n, the expected running time of `QuickSort` is$O(n \log n)$.*

Recall that a random variable is a function that maps every element in the sample space to a real number. In the case of rolling a die, the real number (or value of the point in the sample space) would be the number on the top of the die. Here the sample space is the set of all possible choices of pivots, and an example for a random variable can be the running time of `Quicksort` on a specific input I.

Denote by z_i the i-th element in the sorted array. For each i, j, we define a random variable $X_{i,j}(\sigma)$ to be the number of times z_i and z_j are compared for a given series of pivot choices σ. What are the possible values for $X_{i,j}(\sigma)$? It can be 0 if z_i and z_j are not compared. Note that all comparisons are with the pivot, and that the pivot is not included in the elements of the arrays in the recursive calls. If z_i and z_j are compared, consider the first time that this happens: one of them must be the pivot at this stage and this pivot is excluded from the subarrays that recursive calls operate on. Thus, *no two elements are compared twice*. Therefore, $X_{i,j}(\sigma) \in \{0, 1\}$.

Our goal is to compute the expected number of comparisons that `Quicksort` makes. Recall the definition of expectation:

$$\mathbb{E}[X] = \sum_{\sigma} \mathbb{P}[\sigma] X(\sigma) = \sum_{k} k \mathbb{P}[X = k]$$

Unfortunately for us, this definition does not really give us any clue about how we can actually go about computing the expectation of the complicated random variable that we have at hand, i.e. the number of comparisons made by `Quicksort`. Can you imagine trying to figure out the probability that `Quicksort` actually performs exactly k comparisons? We don't have the foggiest idea how we might do that. If you do, please let us know!

Luckily for us, there is way around this. An important property of expectation is **linearity of expectation**. For any random variables $X_1, \cdots, X_n$:

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i]$$

This is a really simple to state, but amazingly useful property that makes the computation of expectations of seemingly complicated random variables much much easier. It is worth internalizing this technique that we are about to apply for analyzing the expected number of comparisons that `Quicksort` performs.

We start with computing the expected value of $X_{i,j}$. These variables are indicator random variables, which take the value 1 if some event happens, and 0 otherwise. The expected value is:

$$\begin{aligned} \mathbb{E}[X_{i,j}] &= \mathbb{P}[X_{i,j} = 1] \cdot 1 + \mathbb{P}[X_{i,j} = 0] \cdot 0 && \text{(definition of expectation)} \\ &= \mathbb{P}[X_{i,j} = 1] \end{aligned}$$

Let $C(\sigma)$ be the total number of comparisons made by `Quicksort` for a given set of pivot choices σ:

$$C(\sigma) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{i,j}(\sigma)$$

We wish to compute $\mathbb{E}[C]$ to get the expected number of comparisons made by `Quicksort` for an input array of size n.

$$
\begin{aligned}
\mathbb{E}[C] &= \mathbb{E}\left[\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{i,j}(\sigma)\right] && \text{(definition of } C\text{)}\\
&= \sum_{i=1}^{n}\mathbb{E}\left[\sum_{j=i+1}^{n} X_{i,j}(\sigma)\right] && \text{(linearity of expectation on first sum)}\\
&= \sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbb{E}[X_{i,j}(\sigma)] && \text{(linearity of expectation on second sum)}\\
&= \sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbb{P}[z_i, z_j \text{ are compared}] && \text{(definition of expectation on indicator variable)}
\end{aligned}
$$

Now we find $P[z_i, z_j \text{ are compared}]$. Apriori, this seems difficult to do, but it turns out that there is a really elegant way to analyze this. Note that each element in the array(except the pivot) is compared to the pivot at each level of the recurrence. To analyze $P[z_i, z_j \text{ are compared}]$, examine the portion of the array $[z_i, \cdots, z_j]$. After the array is split using a pivot from $[z_i, \cdots, z_j]$, z_i and z_j can no longer be compared. Hence, z_i and z_j are compared only when from the portion of the array $[z_i, \cdots, z_j]$, either z_i or z_j is the first one picked as the pivot. So,

$$
\begin{aligned}
\mathbb{P}[z_i, z_j \text{ are compared}] &= \mathbb{P}[z_i \text{ or } z_j \text{ is the first pivot picked from } [z_i, \cdots, z_j]]\\
& && \text{(argued above)}\\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} && \text{(picking from a set uniformly)}\\
&= \frac{2}{j-i+1}
\end{aligned}
$$

Make sure you are able to explain the second line in the calculation above.

We return to the expected value of C:

$$
\begin{aligned}
\mathbb{E}[C] &= \sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbb{P}[z_i, z_j \text{ are compared }]\\
&= \sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{2}{j-i+1}
\end{aligned}
$$

Note that for a fixed i,

$$\begin{aligned}\sum_{j=i+1}^{n} \frac{1}{j-i+1} &= \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-i+1} \\ &\leq \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\end{aligned}$$

And using $\sum_{k=2}^{n} \frac{1}{k} \leq \ln n$, we get that

$$\begin{aligned}\mathbb{E}[C] &= \mathbb{E}\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{i,j}(\sigma)\right] \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &\leq 2n \ln n\end{aligned}$$

Thus, the expected number of comparisons made by `Quicksort` is no greater than $2n \ln n = O(n \log n)$. To complete the proof, we have to show that the running time is dominated by the number of comparisons. Note that in each recursive call to `Quicksort` on an array of size k, the algorithm performs $k-1$ comparisons in order to split the array, and the amount of work done is $O(k)$. In addition, `Quicksort` will be called on single-element arrays at most once for each element in the original array, so the total running time of `Quicksort` $isO(C+n)$. In conclusion, the expected running time of `Quicksort` on worst-case input is $O(nlogn)$.

1.2 Alternative Proof

Here we provide an alternative method for bounding the expected number of comparisons. Let $T(n)$ be the expected number of comparisons performed by `Quicksort` on an input of size n. In general, if the pivot is chosen to be the i-th order statistic of the input array (i.e. the ith largest element),

$$T(n) = n - 1 + T(i-1) + T(n-i)$$

where we define $T(0) = 0$. Each of the n possible choices of i are equally likely. Thus, the expected number of comparisons is:

$$\begin{aligned}T(n) &= n - 1 + \frac{1}{n} \sum_{i=1}^{n} (T(i-1) + T(n-i)) \\ &= n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} T(i) \qquad \text{(Each } T(i) \text{ shows up twice in the sum)}\end{aligned}$$

Did you catch this minor detail: why does the second summation go from 1 through $n-1$, while on the previous line, the summation goes from 1 through n?

Continuing the calculation, we will use two facts:

1. $\sum_{i=1}^{n-1} f(i) \leq \int_1^n f(x)dx$ for an increasing function f. (How would you prove this? This gives us a way to upper-bound discrete sums by integrals of continuous functions – a nice trick to have in your toolkit that we will use below. If you understand this, what is the corresponding statement to lower-bound a discrete sum by an integral? What if f was a decreasing function of n? It is good to get into the habit of asking yourself such questions to test your understanding.)
2. $\int 2x \ln x dx = x^2 \ln x - \frac{x^2}{2} + C$

Now we show that $T(n) \leq 2n \ln n$ by (strong) induction.

Inductive Hypothesis. $T(i) \leq 2i \ln i$.

Base case (i = 1) An array of size 1 requires no comparisons. Thus, $T(1) = 0$ and the inductive hypothis is true for $i = 1$.

Inductive step. We will show that if the inductive hypothesis is true for all $i \leq k-1$ then the inductive hypothesis is also true for $i = k$.

Let's bound $T(k)$

$$\begin{aligned}
T(k) &= k - 1 + \frac{2}{k}\sum_{i=1}^{k-1} T(i) && \text{(by definition)} \\
&\leq k - 1 + \frac{2}{k}\sum_{i=1}^{k-1} 2i \ln i && \text{(using our strong inductive hypothesis)} \\
&\leq k - 1 + \frac{2}{k}\int_1^k (2x \ln x)dx && \text{(Using fact 1 above)} \\
&= k - 1 + \frac{2}{k}\left[k^2 \ln k - \frac{k^2}{2} + \frac{1}{2}\right] && \text{(Using fact 2 above)} \\
&= 2k \ln k + k - 1 - k + \frac{1}{k} && \text{(simplify)} \\
&= 2k \ln k - 1 + \frac{1}{k} \\
&\leq 2k \ln k
\end{aligned}$$

Thus the inductive hypothesis is true for $i = k$. This establishes the inductive step.

Conclusion By induction, we have proved that $T(i) \leq 2i \ln i$ for all i. Hence $T(n) \leq 2n \ln n$. This concludes the proof.

2 Lower Bounds for Comparison-Based Sorting Algorithms

This is not covered in this course, however, its an interesting exercise to follow. If you'd like to learn more as to why the fastest possible comparison-based sorting algorithm must be $\Omega(n \log n)$ (eg, must take at least $n \log n$ steps in the worst-case), see these http://www.cs.cmu.edu/ avrim/

3 Counting Sort

Recall that our sorting lower bounds applied to the class of algorithms that can only evaluate the values being sorted via *comparison queries*, namely via asking whether a given element is greater than, less than, or equal to some other element. For such algorithms, it's been proven that any correct algorithm (even a randomized one!) will require $\Omega(n \log n)$ such queries on some input. As with any lower-bound that we prove in a restricted model, it is fruitful to ask "is it possible to have an algorithm that does not fall in this class, and hence is not subject to the lowerbound?" For sorting, the answer is "yes".

We start by looking at a very simple non-comparison-based sorting algorithm called Counting Sort. Since it is not comparison-based, it is not restricted by the $\Omega(n \log n)$ lower bound for sorting. For a given input of n objects, each with a corresponding key (or value) in the range $\{0, 1, \cdots, r-1\}$, Counting Sort will sort the objects by their keys:

1. Create an array A of r buckets where each bucket contains a linked list.
2. For each element in the input array with key k, concatenate the element to the end of the linked list $A[k]$.
3. Concatenate all the linked lists: $A[0], \cdots, A[r-1]$.

The algorithm correctly sorts the n elements by their keys because the elements are placed into buckets by key where bucket i (containing elements with key $= i$) will come before bucket j (containing elements with key $= j$) in A for $i < j$. Therefore, when the algorithm concatenates the buckets, all elements with key $= i$ will come before elements with key $= j$. The worst case run time of Counting sort is $O(n + r)$ since it performs $O(1)$ passes over the n input elements and $O(1)$ passes over the r buckets of A. This is great if the size of the range r is small, but we pay dearly in the running time (and space) if r is large. The algorithm we discuss in the next section builds on Counting Sort and fixes this issue (that Counting Sort behaves poorly if the range of values r is very large).

An important property (which we will use in the next section) is that the algorithm described above is *stable*: If two input elements x, y have the same key, and x appears before y in the input array, x will appear before y in the output.

4 Radix Sort

Radix Sort is another non-comparison-based sorting algorithm that will use Counting Sort as a subroutine. Assuming that the input array contains d-digit numbers where each digit ranges from 0 to $r-1$, Radix Sort sorts the array digit-by-digit (or field-by-field for non-numerical inputs). The algorithm works on input array A as follows:

1. For $j = 1, \cdots, d$:
2. Apply Counting Sort to A using the jth digit as the key.

Note that we refer to the least significant digit as the first digit. Hence, the algorithm calls Counting Sort first using the least significant digit as the key, then again using the second least significant digit, until the most significant digit.

We will show that Radix Sort correctly sorts an input list of n numbers via induction on the iterations of the loop.

Inductive Hypothesis: At the end of the j^{th} iteration, the elements in A are sorted when considering only the j least significant digits of each element.

Base case: ($j = 1$) Radix Sort correctly sorts the numbers by the first digit as it uses Counting Sort to sort the numbers using the least significant digit as a key. (Note: we could have used $j = 0$as the base case as well).

/Inductive case: We will prove that, if the inductive hypothesis for $j = k-1$ is true, then the inductive hypothesis for $j = k$ is true as well. By the inductive hypothesis for $j = k-1$, by the end of iteration $k-1$, the input numbers have been sorted by the $k-1$ least significant digits. After we run Counting Sort on the elements using digit k as the key, the numbers are sorted by their j^{th} digit. Since Counting Sort is stable, the elements in each bucket keep their original order, and by the induction hypothesis, they are ordered by their $k-1$ least significant digits. Since **the elements are ordered first by their k^{th} digit, and then by their $k-1$ least significant digits**, we conclude that they are ordered by their k least significant digits.

To provide more detail for the bolded text in that last sentence, we could consider any two numbers x and y in the input and separately consider two cases as we did in lecture: case (i) where x and y have different kth least significant digits, and case (ii) where x and y have identical kth least significant digits. In both cases, we could establish that, in the output,x and y are ordered by their least k significant digits. Since this is true for every pair of numbers x and y in the input, this means that the output of this iteration is correctly sorted by the k least significant digits.

By induction, the inductive hypothesis is true for all $j \in \{1, \cdots, d\}$. Applying the inductive hypothesis for $j = d$, we conclude that at the end of iteration d, the numbers are in sorted order (since the input consists of d-digit numbers). In other words, Radix Sort correctly sorts the input.

The worst case running time of Radix Sort is $O(d(n+r))$ since we are calling Counting Sort on n elements with r possible keys once for each digit in the input numbers. If $r = O(n)$ and $d = O(1)$, then this takes $O(n)$ time.

As we did in lecture, we could consider varying the base r in which we write our numbers and running radix sort (with the Counting Sort in the inner loop using r buckets). Now the maximum number of digits is a function of the maximum size of the numbers in the input and our choice of base r. If we have an input consisting of n numbers of value at most M, then the number of digits $d = \lfloor log_r(M) \rfloor + 1$. (verify this yourself!) Thus the running time for Radix Sort with base r is $O(d(n+r))$ which is $O((\lfloor \log_r(M) \rfloor + 1) \cdot (n+r))$.

How should we choose r? One reasonable choice is $r = n$ to balance the two values n and r in the term $(n+r)$ in the expression for the running time. For this choice of r, the running time for Radix sort is $O(n \cdot (\lfloor \log_r(M) \rfloor + 1))$. (Why didn't we drop the $+1$ term in the expression for the asymptotic running time?) Note that we might make a different choice of r if it was also important to optimize the space required to run Radix Sort.

So the running time of Radix Sort (in terms of n) depends on how large the numbers in the input are as a function of n. If the upper bound $M \leq n^c$ for some constant c, then the running time is $O(n)$. In this regime, Radix Sort beats MergeSort and Quicksort. On the other hand, if $M = 2^n$, then the running time is $O(n^2/\log n)$. In this regime, we would not use Radix Sort.