Average-Case Analysis and Randomization

Quicksort Revisited

```
QuickSort(A, l, r)
if r ≤ l then return
p = FindPivot(A, l, r)
m = Partition(A, l, r, p)
QuickSort(A, l, m)
QuickSort(A, l, m+1, r)
```

If $r-l+1=n$, $m-r+1=n_1$, and $r-m=n_2$, then the running time is given by the recurrence

$$T(n) = T(n_1) + T(n_2) + \Theta(n).$$

Since the partition ensures $n_1 \geq 1$ and $n_2 \geq 1$, the algorithm terminates. The crux is the line highlighted in green. If we choose p to be the median of $A[l, r]$, then $n_1 = n_2 = n/2$. Since we can use linear-time selection to find the median, we thus obtain a variant of QuickSort with worst-case running time

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n).$$

In practice, this algorithm is slow – linear-time selection is a too heavy gun for the simple task of finding a pivot. What about the optimistic approach?

```
FindPivot(A, l, r)
return A[l]
```

We can't find the pivot faster than that and, if $A[l]$ happens to be at least close to the median in each step, we still get a $\Theta(n \lg n)$ running time. But we could equally well be unlucky and $A[l] = \min A[l..r]$ every time, in which case we get

$$T(n) = T(1) + T(n-1) + \Theta(n) = \Theta(n^2).$$

We want to prove that the average-case running time of this algorithm is $\Theta(n \lg n)$. Recall that this is the average running time over all possible inputs of size n. Since there are infinitely many such inputs, this average is a little hard to compute. To overcome this problem, we group the inputs into a finite number of classes of inputs for which the algorithm behaves exactly the same and we determine the expected running time for a random input that is equally likely to come from any of these classes.

For quicksort, we observe that it behaves exactly the same for inputs 4, 2, 3, 1 and 11, 5, 9, 2 because both inputs start with the largest element, followed by the second smallest, and so on, and QuickSort uses only comparisons, not their values, to determine the sorted order. Thus, QuickSort only cares which permutation of the input elements is given as the input and there are "only" $n!$ different permutations, a finite number.

So our goal is to prove that the running time of QuickSort is $O(n \lg n)$ for a uniformly random input permutation. This can be shown for the variant of QuickSort we discussed before, but this is somewhat tricky because, even though the initial input is a uniform random permutation, this is not quite true for the inputs of recursive calls. To make the algorithm easier to analyze, we switch from Hoare's to Lomuto's partition algorithm and ensure that the pivot is not included in either of the two recursive calls:

```
QuickSort(A, l, r)
if r ≤ l then return
m = Partition(A, l, r)     // No need to select a
QuickSort(A, l, m-1)       // pivot because Partition
QuickSort(A, m+1, r)       // simply uses A[r] as
                           // pivot
```

```
Partition(A, l, r)
i = l - 1
for j = l to r - 1 do
    if A[j] ≤ A[r] then
        i = i + 1
        swap A[i] and A[j]
swap A[i+1] and A[r]
return i+1
```

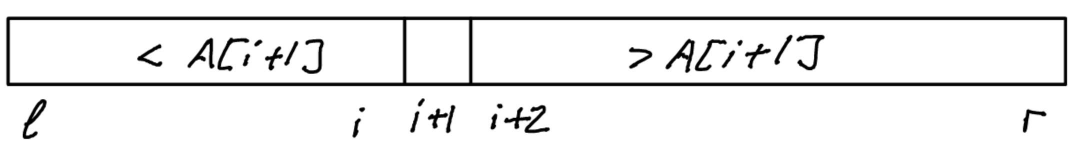
Before the last two lines, the partition algorithm maintains the invariant that

(i) The elements in $A[\ell..i]$ are less than $A[r]$
(ii) The elements in $A[i+1..j-1]$ are greater than $A[r]$
(iii) The elements in $A[j..r-1]$ still need to be inspected.

Indeed, this invariant holds initially because $i = \ell - 1$ and $j = \ell$, and it is not hard to show that each iteration of the for-loop maintains this invariant. Thus, once the for-loop exits, we have:

(a) The elements in $A[\ell..i]$ are less than $A[r]$
(b) The elements in $A[i+1..r-1]$ are greater than $A[r]$

Swapping $A[i+1]$ and $A[r]$ at this point thus produces the configuration



Thus, by recursively sorting $A[\ell..i]$ and $A[i+2..r]$ as QuickSort does, we end up sorting $A[\ell..r]$. Now back to the analysis:

Lemma: Let X be the number of comparisons QuickSort performs. Then its running time is $O(n+X)$.

Proof: The running time of Partition is proportional to the number of comparisons it performs. Apart from calling Partition, each invocation takes constant time. The number of invocations is given by $I(n) = 1 + I(n_1) + I(n_2)$, where $n_1 \geq 0$, $n_2 \geq 0$ and $n_1 + n_2 = n-1$. This is easily shown to solve to $I(n) \leq 2n-1$. $\square$

By this lemma, it suffices to prove that $E[X] \in O(n \lg n)$.

Let $a_1 < a_2 < \dots < a_n$ be the input elements. Note that they are not necessarily stored in this order in the input array. Let

$$X_{ij} = \begin{cases} 1 & \text{if } a_i \text{ and } a_j \text{ are compared at least once} \\ 0 & \text{otherwise} \end{cases}$$

By the following lemma, we have $X = \sum_{i<j} X_{ij}$ and thus $E[X] = \sum_{i<j} E[X_{ij}]$.

Lemma: Two input elements a_i and a_j are compared at most once.

Proof: a_i and a_j are compared only if one of them is chosen as a pivot. Let QuickSort(A, l, r) be the first recursive call that chooses one of a_i and a_j as a pivot, say a_i. Then a_i does not belong to the input of any descendant invocation of QuickSort(A, ℓ, r), so even if one of them chooses a_j as a pivot, this cannot cause a comparison between a_i and a_j. $\square$

It remains to bound $E[X_{ij}]$ for all $i<j$. To do so, we first need to prove that the input to every invocation QuickSort(A, ℓ, r) is a uniformly random permutation, assuming the initial input array $A[1..n]$ is.

Lemma: If $A[1..n]$ is a uniformly random permutation, then so is the input of every recursive call QuickSort$(A, \ell, r]$ is the recursion tree of QuickSort$(A, 1, n)$.

Proof: By induction on the number of ancestor invocations of QuickSort(A, ℓ, r). If there are no ancestor invocations, the claim holds because initially $A[1..n]$ is a uniformly random permutation. Otherwise, let QuickSort(A, ℓ', r') be the parent invocation of QuickSort(A, ℓ, r). By the inductive hypothesis, $A[\ell'..r']$ is a uniformly random permutation when we call QuickSort(A, ℓ', r'). Assume w.l.o.g. that QuickSort(A, ℓ, r) is the left child invocation of QuickSort(A, ℓ', r'), that is, $\ell = \ell'$ and $n_1 = r - \ell + 1$ is the number of elements in $A[\ell'..r']$ no greater than $A[r']$. Let us mark the elements in $A[\ell'..r'-1]$ with a $-$ or $+$ depending on whether they are less than or greater than $A[r']$. There are exactly n_1 $-$'s. For each possible pattern of $+$'s and $-$'s, Position $[\ell'..r']$

produces a fixed permutation of $A[\ell'..r']$ because it uses only comparisons with $A[r']$ to produce this partition. In particular, permuting the "$-$"-elements does not change the permutation of $A[\ell'..r']$ produced by Partition(A, ℓ', r'). Since, $A[\ell'..r']$ is a uniformly random permutation before calling Partition(A, ℓ', r'), so is the subsequence of "$-$"-elements. Since a fixed permutation of a uniformly random permutation is itself a uniformly random permutation, this shows that $A[\ell..r]$ is a uniformly random permutation once Partition(A, ℓ', r') finishes. $\square$

Now we are ready to bound $E[X]$. In particular, we prove the following lemma:

Lemma: $E[X_{ij}] = \frac{2}{j-i+1}$.

Proof: Let QuickSort(A, ℓ, r) be the first invocation that chooses one of $a_i, \dots, a_j$ as a pivot. Then $a_i, a_j \in A[\ell..r]$ and $X_{ij} = 1$ iff one of a_i and a_j is the chosen pivot. (If one of them is chosen as the pivot, then $X_{ij} = 1$. Otherwise, the pivot is some a_k with $i < k < j$. This implies that a_i and a_j belong to the inputs of different child invocations of QuickSort(A, ℓ, r) and are never compared, that is, $X_{ij} = 0$.) Now, the pivot is just $A[r]$. Since $A[\ell..r]$ is a uniformly random permutation, any element in $\{a_i, \dots, a_j\}$ is equally likely to be $A[r]$, that is, the probability that a_i is chosen as the pivot is $1/(j-i+1)$. The probability of choosing a_j is also $1/(j-i+1)$, by the same argument. Thus, the probability of choosing one of the two is $\frac{2}{j-i+1}$. This gives

$$E[X_{ij}] = 1 \cdot \Pr[a_i \text{ or } a_j \text{ are chosen as pivot}] + 0 \cdot \Pr[a_i \text{ and } a_j \text{ are not chosen as pivot}]$$
$$= \frac{2}{j-i+1}. \qquad \square$$

This gives

$$E[X] = \sum_{i<j} E[X_{ij}]$$
$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$
$$< 2 \sum_{i=1}^{n-1} \sum_{j=1}^{n} \frac{1}{j}$$
$$= 2(n-1) H_n,$$

where $H_n = \sum_{j=1}^{n} \frac{1}{j}$ is then nth Harmonic number.

By the following lemma, this gives $E[X] \in O(n \lg n)$, that is, the expected running time of QuickSort is $O(n \lg n)$.

Lemma: $\ln(n+1) \leq H_n < \ln n + 1$

Proof:
$$H_n = \int_1^{n+1} \frac{1}{\lfloor x \rfloor} dx > \int_1^{n+1} \frac{1}{x} dx = \ln x \Big|_1^{n+1} = \ln(n+1).$$
$$H_n = 1 + \int_1^{n} \frac{1}{\lceil x \rceil} dx < 1 + \int_1^{n} \frac{1}{x} dx = 1 + \ln x \Big|_1^{n} = 1 + \ln n.$$
$\square$

A problem with average case analysis is that we assume every input is equally likely. This may or may not be true for the particular application where we want to use a given algorithm. For example, one strategy to integrate a small number of new records into a sorted database table is to concatenate the sorted table with the sequence of new records and then sort the resulting sequence. In this case, the sequences we sort are almost sorted, except for the small number of new records, and QuickSort always takes $\Omega(n^2)$ time on such inputs.

A better strategy is to let the algorithm make random choices - we call this a randomized algorithm - and take the average running time over all possible random choices. If we do this, we no longer assume a probability distribution over the set of all possible inputs, which may or may not be correct, but we base our analysis on a known probability distribution over the random choices the algorithm makes.

In the case of QuickSort, the change is trivial. The only part of our analysis that used the assumption that the input is a uniformly random permutation was where we argued that each of the elements in $A[l..r]$ is equally likely to be stored in $A[r]$, that is, it is equally likely to be chosen as a pivot. To turn this assumption into a certainty, we only need to choose an element from $A[l..r]$ uniformly at random and swap it with $A[r]$:

```
QuickSort(A, l, r)
  if r ≤ l then return
  j = random(l, r)  // Pick a random integer between l and r
  swap A[j] <=> A[r]
  m = Partition(A, l, r)
  QuickSort(A, l, m-1)
  QuickSort(A, m+1, r)
```

The same analysis as before now shows the following lemma.

Lemma: For any input $A[1..n]$, the expected running time of QuickSort$(A, 1, n)$ is in $O(n \lg n)$.

With a little more effort one can show that the probability that the running time of QuickSort$(A, 1, n)$ exceeds $c n \lg n$, for some constant $c > 1$, is $\frac{1}{n^d}$, where d is a function of c. Thus, essentially you will never see randomized QuickSort take more than $O(n \lg n)$ time even though it is theoretically possible.

Randomized Selection

Given that random pivot selection works well for QuickSort, it should also work well for linear-time selection. Let's try to prove it. Here's the algorithm:

```
Select (A, l, r, k)
 if r ≤ l then return A[l]
 j = random (l, r)
 Swap A[j] <=> A[r]
 m = Partition (A, l, r)
 if m - l + 1 = k then return A[m]
 else if k < m - l + 1 then return Select (A, l, m-1, k)
 else return Select (A, m+1, r, k - (m-l+1))
```

Again, we use Lomuto's partition algorithm, even though that's not important if we pick the pivot uniformly at random.

Lemma: The expected running time of the randomized Select algorithm is $O(n)$.

Proof: If we choose the ith smallest element as pivot, then we have $T(n) \le cn + \max(T(i-1), T(n-i))$. We choose this element with probability $\frac{1}{n}$. Thus,

$$E[T(n)] \le cn + \sum_{i=1}^{n} \frac{1}{n} \max(E[T(i-1)], E[T(n-i)])$$

$$\le cn + \frac{2}{n} \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} E[T(i)]$$

Now we claim that $E[T(n)] \le dn$, for some $d > 0$. For $n < 12$, this is true. For $n \ge 12$, we obtain

$$E[T(n)] \le cn + \frac{2}{n} \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} di$$

$$= cn + \frac{2}{n} \left(\sum_{i=1}^{n-1} di - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor - 1} di \right)$$

$$= cn + \frac{2d}{n}\left(\frac{n(n-1)}{2} - \frac{\lfloor\frac{n}{2}\rfloor(\lfloor\frac{n}{2}\rfloor-1)}{2}\right)$$

$$\leq cn + \frac{d}{n}\left(n(n-1) - \left(\frac{n}{2}-1\right)\left(\frac{n}{2}-2\right)\right)$$

$$= cn + \frac{d}{n}\left(n^2 - n - \frac{n^2}{4} + \frac{3n}{2} - 2\right)$$

$$\leq cn + \frac{3dn}{4} + \frac{3d}{2}$$

$$\leq cn + \frac{7dn}{8} \quad \left(\text{because } \frac{n}{8} \geq \frac{3}{2}\right)$$

$$\leq dn \quad \text{for all } d \geq 8c. \qquad \square$$

Sorting in Linear Time

Using comparisons only, we cannot beat the $n \lg n$ sorting bound achieved for example by QuickSort and MergeSort, but what if we are willing to use other operations and the input is random? In particular, assume we know every input element comes from the interval $[\ell, u]$ and is drawn uniformly at random from this interval. Then the following lemma gives us an algorithm that sorts the input in $O(n^2)$ time in the worst case and $O(n)$ time on average.

Let us define n intervals $I_1, I_2, \dots, I_n$, where $I_j = [\ell + \frac{j-1}{n}(u-\ell), \ell + \frac{j}{n}(u-\ell)]$, and let X_j be the number of input elements that fall into interval I_j.

Lemma: $E[X_j^2] < 2.$

Proof: Let $Y_i = 1$ if the ith element belongs to I_j and $Y_i = 0$ otherwise. Then $X_j = \sum_{i=1}^{n} Y_i$, so

$$
\begin{aligned}
E[X_j^2] &= E\left[\left(\sum_{i=1}^{n} Y_i\right)^2\right] \\
&= E\left[\sum_{i=1}^{n}\sum_{\substack{k=1\\k\neq i}}^{n} Y_i Y_k + \sum_{i=1}^{n} Y_i^2\right] \\
&= \sum_{i=1}^{n}\sum_{\substack{k=1\\k\neq i}}^{n} E[Y_i Y_k] + \sum_{i=1}^{n} E[Y_i^2] \\
&= \sum_{i=1}^{n}\sum_{\substack{k=1\\k\neq i}}^{n} E[Y_i]E[Y_k] + \sum_{i=1}^{n} E[Y_i^2]
\end{aligned}
$$

by linearity of expectation and because Y_i and Y_k are independent for $k \neq i$. Now, the ith input element belongs to I_j with probability $\frac{1}{n}$. When this happens $Y_i = 1$. Otherwise, $Y_i = 0$. Thus, $E[Y_i] = \frac{1}{n} \cdot 1 + (1-\frac{1}{n}) \cdot 0 = \frac{1}{n}$ and $E[Y_i^2] = \frac{1}{2} \cdot 1^2 + (1-\frac{1}{n}) \cdot 0^2 = \frac{1}{n}$. This gives

$$
\begin{aligned}
E[X_j^2] &= \sum_{i=1}^{n}\sum_{\substack{k=1\\k\neq i}}^{n} \frac{1}{n^2} + \sum_{i=1}^{n} \frac{1}{n} \\
&= \frac{n(n-1)}{n^2} + 1 \\
&< 2.
\end{aligned}
$$

□

The sorting algorithm is simple now: We position the input elements based on the interval I_j they belong to. This takes $O(n)$ time. Then we sort the elements in each interval using insertion sort. By the previous lemma, the cost per interval is constant on average. Since we have n intervals, the average-case cost of sorting the elements in all intervals is $O(n)$. Here's the algorithm's pseudo-code.

```
BucketSort(A, l, u)
Create n empty buckets B_1, ..., B_n represented as
linked lists.
for i=1 to n do
    j = ⌈n · (A[i]-l)/(u-l)⌉
    Append A[i] to B_j
for j=1 to n do
    InsertionSort(B_j)
Concatenate B_1, ..., B_n and return the result
```