# CS 31: Algorithms (Spring 2019): Lecture 9

Date: 23rd April, 2019

Topic: Randomized Algorithms 1: Frievald's Algorithm, Quicksort

*Disclaimer: These notes have not gone through scrutiny and in all probability contain errors.*
*Please notify errors on Piazza/by email to deeparnab@dartmouth.edu.*

---

Randomized algorithms are algorithms which have an extra resource available to them: they have access to a stream of *independent* fair random coins. More pertinently, we assume the algorithm has access to a subroutine `rand()`, and each time the algorithm calls it, it obtains a bit which is $0$ or a $1$, equally likely. Furthermore any two calls to `rand()` produces answers which are independent of each other.

There are two ways in which randomness can affect the performance of traditional algorithms. These two ways lead to two classes of randomized algorithms.

**Monte Carlo Algorithms.** A randomized algorithm $\mathcal{A}$ on an input $I$ could run in some *fixed, deterministic* time $T_{\mathcal{A}}(I)$, but return a solution $\mathcal{A}(I)$ which *could* be ***wrong***. However, the probability that the algorithm is wrong (the probability taken over the randomness generated by the algorithm's various calls to `rand`()) is "small". Formally, for our lectures let's assert that

$$\mathbf{Pr}[\mathcal{A}(I) \text{ is wrong}] \leq 2^{-100}$$

These are called *Monte Carlo* randomized algorithms.

**Las Vegas Algorithms.** A randomized algorithm $\mathcal{A}$ on input $I$ could *always* return the correct solution $\mathcal{A}(I)$, however, the running time $T_{\mathcal{A}}(I)$ is a *random variable*. In particular, if one is unlucky the algorithm may just run for ever. For these algorithms, what is more of interest is the *expected* running time. In particular, on input $I$ we care for $\mathbf{Exp}[T_{\mathcal{A}}(I)]$. Indeed, the runtime as a function of the size is defined as

$$T_{\mathcal{A}}(n) := \max_{I:|I|\leq n} \mathbf{Exp}[T_{\mathcal{A}}(I)]$$

In this lecture, we will see examples of both types of randomized algorithms.

## 1 Checking Matrix Multiplication: Frievald's Algorthm

CHECKING MATRIX MULTIPLICATION

**Input:** Three $n \times n$ matrices $A, B, C$.

**Output:** Decide whether $A \cdot B = C$ or not?

**Size:** $n$.

Recall that the product of two $n \times n$ matrices $A, B$ is another $n \times n$ matrix $C$ defined as follows

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}, \quad \forall 1 \leq i, j \leq n$$

Naively, two matrices need $n^3$ time to be multiplied; each product as above needs $O(n)$ multiplications, and there are $n^2$ pairs $(i, j)$. If you have been looking at the supplemental problems, then you may have seen a faster algorithm to multiply two matrices using Divide-and-Conquer. The current record is $O(n^{2.377})$ time algorithms discovered in 2015. So the obvious algorithm of "multiply $A$ and $B$ and check whether it is equal to $C$" runs in at best $O(n^{2.377})$ time.

We now see an extremely simple Monte-Carlo algorithm with one-sided error which runs in quadratic time.

The main observation is that if $A \cdot B = C$, then for **any** $n$-dimensional vector $\mathbf{r} = (r_1, \ldots, r_n)$, we must have $(A \cdot B) \cdot \mathbf{r} = C \cdot \mathbf{r}$. Observe that calculating both the LHS and RHS takes quadratic time. To see this for the LHS, we need to use the associativity of multiplication: $(A \cdot B) \cdot \mathbf{r} = A \cdot (B \cdot \mathbf{r})$, and $(B \cdot \mathbf{r})$ is also an $n$-dimensional vector.

On the other hand, even when $A \cdot B \neq C$, there may be some $\mathbf{r}$ for which $(A \cdot B) \cdot \mathbf{r} = C \cdot \mathbf{r}$. For instance, suppose $M = A \cdot B$, and $C \equiv M$ except $C[1, 1] \neq M[1, 1]$. Then for any $\mathbf{r}$ with $r_1 = 0$, we will indeed have $(A \cdot B) \cdot \mathbf{r} = C \cdot \mathbf{r}$. Check this.

On the other hand, as we show below, if $\mathbf{r}$ is chosen *randomly*, more precisely, if $r_i \in \{0, 1\}$ is chosen uniformly at random, then the algorithm will "catch the discrepancy" with high probability.

```
procedure FRIEVALD($A, B, C$):
    ▷ Checks whether $A \cdot B = C$ or not.
    for $i = 1$ to $k$ do:
        Generate $n$ independent random variables $r_i \in \{0, 1\}$ calling rand().
        Let $\mathbf{r} \leftarrow (r_1, \ldots, r_n)$.
        Compute $u \leftarrow (A \cdot (B \cdot \mathbf{r}))$.
        Compute $v \leftarrow C \cdot \mathbf{r}$.
        return REJECT if $u \neq v$.
    return ACCEPT ▷ None of the $k$ tries above reject.
```

**Remark:** *The running time of* FRIEVALD *is $O(kn^2)$. So for a constant $k$, it runs in $O(n^2)$ time.*

**Theorem 1.** The Algorithm FRIEVALD returns a wrong answer with probability $\leq \frac{1}{2^k}$.

*Proof.* Let us first check that if $A \cdot B = C$, then the algorithm *never* makes an error (no matter what rand() returns.) This is because for any $\mathbf{r}$ we have $(A \cdot B) \cdot \mathbf{r} = C \cdot \mathbf{r}$. Therefore, FRIEVALD will return ACCEPT. The interesting part is the other direction.

Now, suppose $A \cdot B \neq C$. Let $D := A \cdot B - C$; $D$ must have at least one non-zero entry, and in particular, at least one non-zero row. Suppose this row $\mathbf{w}$. Now observe that each for-loop of FRIEVALD rejects is the probability $\mathbf{Pr}[D \cdot \mathbf{r} \neq 0]$. In particular, this is at least $\mathbf{Pr}[\mathbf{w} \cdot \mathbf{r} \neq 0]$. The next claim shows that this probability is at least $1/2$; therefore, the probability none of the $k$ for-loops rejects, since we are each time using independent random coins, is $\geq 1 - 1/2^k$.

**Claim 1.** Let $\mathbf{w}$ be *any* non-zero $n$-dimensional vector. Let $\mathbf{r}$ be a random $n$-dimensional vector with $r_i \in \{0, 1\}$ with probability $1/2$. Then,

$$\mathbf{Pr}[\mathbf{w} \cdot \mathbf{r} = 0] \ \leq \ 1/2$$

*Proof.* Since $\mathbf{w} \neq 0$, we know one of its coordinates is $\neq 0$. Without loss of generality let us assume it is $w_1$. Then, let's write $\mathbf{w} \cdot \mathbf{r}$ as

$$\mathbf{w} \cdot \mathbf{r} = \sum_{i=1}^{n} w_i r_i = (w_1 r_1) + \sum_{i=2}^{n} w_i r_i$$

Let $X$ be the random variable $w_1 r_1$ and let $Y = \sum_{i=2}^{n} w_i r_i$. The *crucial* thing to note is that $X$ and $Y$ are independent random variable. Let $Z$ be the random variable $\mathbf{w} \cdot \mathbf{r}$. Thus, $Z = X + Y$.

$$\begin{aligned}
\mathbf{Pr}[Z = 0] = \quad & \mathbf{Pr}[Z = 0 | Y = 0] \cdot \mathbf{Pr}[Y = 0] \ + \ \mathbf{Pr}[Z = 0 \mid Y \neq 0] \cdot \mathbf{Pr}[Y \neq 0] \\
\leq \quad & \mathbf{Pr}[X = 0 | Y = 0] \cdot \mathbf{Pr}[Y = 0] \ + \ \mathbf{Pr}[X \neq 0 \mid Y \neq 0] \cdot \mathbf{Pr}[Y \neq 0] \\
= \quad & \mathbf{Pr}[X = 0] \cdot \mathbf{Pr}[Y = 0] \ + \ \mathbf{Pr}[X \neq 0] \cdot (1 - \mathbf{Pr}[Y = 0]) \\
= \quad & 1/2, \qquad \text{since } w_1 \neq 0\text{, we have } \mathbf{Pr}[X = 0] = \mathbf{Pr}[r_1 = 0].
\end{aligned}$$

To explain the inequality above, note that $Z = X + Y$. Therefore, given $Y = 0$ we get $Z = 0$ if and only if $X = 0$. Thus, $\mathbf{Pr}[Z = 0 | Y = 0] = \mathbf{Pr}[X = 0 | Y = 0]$. On the other hand, given $Y \neq 0$, we get $Z = 0$ if and only if $X = -Y$. At the very least, we *need* (but it doesn't suffice) $X \neq 0$. Thus, $\mathbf{Pr}[Z = 0 | Y \neq 0] \leq \mathbf{Pr}[X = 0 | Y \neq 0]$. □

□

## 2 QuickSort

Next, we look at QuickSort, an algorithm which many of you may have seen in CS 1. It is another sorting algorithm which is randomized, always returns the sorted list, but whose running time is a random variable whose expectation we will bound.

The main idea behind QuickSort is *pivoting*; using a certain element of the array to break the problem into two and then recursing on the two sides. A very Divide-and-Conquer idea.

More precisely, let $q = A[i]$ be an *arbitrary*[1] element of the list. Given $q$, the list $A[1:n]$ can be divided into 3 lists: $A_1 := \{A[i] : A[i] < q\}$, $A_2 = \{A[i] : A[i] = q\}$, and $A_3 = \{A[i] : A[i] > q\}$. Note this takes one scan of the list, and thus $O(n)$ time.

```
procedure PIVOT($A, q$):
    ▷ A is list of length $n$. Returns two lists $A_1, A_2, A_3$ as desired
    Initially $A_1, A_2, A_3$ are null lists.
    for $i = 1$ to $n$ do:
        if $A[i] < q$ then:
            Append $A[i]$ to $A_1$
        else if $A[i] = q$ then:
            Append $A[i]$ to $A_2$
        else:
            Append $A[i]$ to $A_3$.
    return $(A_1, A_2, A_3)$
```

Now, suppose we recursively sort $A_1, A_3$ and suppose $B_1$ and $B_3$ are the sorted versions. Then note, the sorted order of $A$ is precisely $[B_1, A_2, B_3]$, that is the array $B_1$ followed by the $q$'s followed by $B_2$. Done!

Let's try to write a recurrence inequality and try to solve it then. Suppose the length of array $A_1$ is $n_1$ and length of array $A_3$ is $n_3$. Note that $n_1 + n_3 < n - 1$. To divide, we spend $O(n)$ time (this is the pivoting step), and to conquer, we only need $O(1)$ time. Thus, we get

$$T(n) \leq T(n_1) + T(n_3) + O(n) \tag{1}$$

If, say, both $n_1$ and $n_3$ were $\Theta(n)$, then it is not too hard to show that the solution to the above is $O(n \log n)$ (for instance, if $n_1 = n/3$ and $n_3 = 2n/3$, this was left as an exercise long back). However, what if we are unlucky, and we get $n_1 = 0$ and $n_3 = n - 1$. Then, the solution to the above is $T(n) = O(n^2)$. Not great, since the naive sorting algorithm takes $O(n^2)$ time.

The idea of QuickSort is to choose the pivot $q$ *randomly*! Sure, we could be unlucky and choose a split of $n_1, n_3$ which is not balanced. But it seems we would be devilishly unlucky for this to repeat over-and-over again! Indeed, with this simple random choice, we get an *expected* $O(n \log n)$ behavior.

[1] It will be good to know the difference between arbitrary and random. When we say arbitrary, we don't care which element it is and it may be the worst element for whatever purpose. When we say random, we usually mean uniformly at random among the choices available. In particular for this example, there is indeed a big difference as you will see

```
procedure QuickSort($A$):
    ▷ $A$ is a list of length $n$. Returns a sorted order $B$
    Choose $i \in \{1, 2, \ldots, n\}$ at random.
    $q \leftarrow A[i]$.
    $(A_1, A_2, A_3) \leftarrow$Pivot$(A, q)$.
    $B_1 \leftarrow$QuickSort$(A_1)$.
    $B_3 \leftarrow$QuickSort$(A_3)$.
    return $B_1$ appended with $A_2$ appended with $B_3$.
```

**Theorem 2.** Given any list $A[1:n]$, the algorithm QuickSort sorts it in $O(n \log n)$ expected time.

*Proof.* Let us revisit what we need to argue about. Even when we fix a list $A$, the time taken by the algorithm is a random variable $T(A)$. The expected running time is $\mathbf{Exp}[T(A)]$. What we are interested in is $T(n) = \max_{A:|A|=n} \mathbf{Exp}[T(A)]$.

There are two ways to prove this. One is by writing the "expectation version" of (1) and then doing the math. More precisely, since $q$ is random, the numbers $n_1$ and $n_3$ are random variables and so are $T(n_1)$ and $T(n_3)$. Note that $T(n_1)$ itself is an expectation; the fact that $n_1$ is random makes $T(n_1)$ a random variable. We know what the distribution of the $n_1$'s and the $n_3$'s are, which allows us to finally argue about $T(n)$, and we can indeed write a recurrence formula for $T(n)$. You should try writing this, and you will realize the recurrence inequality you get is not fun to solve.

Rather, what we are going to see is a *beautiful* application of linearity of expectation.

We start by making some observations. First is that the total time taken by the algorithm is dominated by the Pivot subroutine, which itself is dominated by the number of comparisons it makes of the form "Is $A[i] < q$?". Second observe that if two entries of the array $A[i]$ and $A[j]$ are ever compared, they are *never* compared again. Thus, the running time of QuickSort can be upper bounded by the number of comparisons the algorithm makes.

Now suppose $B[1:n]$ is the correct sorted order of $A[1:n]$. Define the *indicator random variable* $X_{ij}$, for $i < j$, which is $1$ if the numbers $B[i]$ and $B[j]$ are *ever* compared in the QuickSort algorithm. We don't just mean the first iteration; we mean anywhere in the full run. From the above two observations, we get

$$T(n) = \mathbf{Exp}\left[\Theta\left(\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{ij}\right)\right]$$

By Linearity of expectation (and the $\Theta$ function), we get

$$T(n) = \Theta\left(\sum_{i=1}^{n}\sum_{j=i+1}^{n} \mathbf{Exp}[X_{ij}]\right) \tag{2}$$

So, all that remains is to argue about $\mathbf{Exp}[X_{ij}]$. Now comes the third and final observation. The numbers $S = \{B[i], B[i+1], \ldots, B[j]\}$ are initially in $A$ (surely). Subsequently, either we choose a $q \notin S$ in which case either all of $S$ goes to $A_1$ or all of $S$ goes to $A_3$. Otherwise, we choose $q \in S$, and in that case in subsequent recursive calls $B[i]$ and $B[j]$ are *separated*. Therefore, the only time $B[i]$ and $B[j]$ are compared if the first time $q$ lands in $S$ it must be either $B[i]$ or $B[j]$. If it is not, then $B[i]$ and $B[j]$ are never compared.

Put differently, we get

$$\mathbf{Pr}[X_{ij} = 1] = \mathbf{Pr}[q \in \{i, j\} \mid q \in S]$$

But $q$ is chosen equally likely among the entries of an array. Thus, given that it falls in $S$, the probability it is either $i$ or $j$ is precisely $2/|S| = 2/(j-i+1)$. If this bothers you, do the calculation using the definition. If $n'$ is the length of the array the first time we get $q \in S$, the the numerator of the conditional probability is $2/n'$ while the denominator is $|S|/n'$.

Putting this in (2), we get

$$T(n) = \Theta\left(\sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1}\right)$$

Changing some variables, we get

$$T(n) = \Theta\left(\sum_{i=1}^{n} \sum_{s=2}^{n-i-1} \frac{2}{s}\right) \leq \Theta\left(\sum_{i=1}^{n} \sum_{s=1}^{n} \frac{1}{s}\right)$$

Now we use the fact that $\sum_{s=1}^{n} 1/s = \Theta(\log n)$. This gives $T(n) = \Theta(n \log n)$. □