Chapter 9

Randomized Algorithms

By Sariel Har-Peled, December 17, 2012[①] Version: 0.26

9.1 Some Probability

Definition 9.1.1. (Informal.) A ***random variable*** is a measurable function from a probability space to (usually) real numbers. It associates a value with each possible atomic event in the probability space.

Definition 9.1.2. The ***conditional probability*** of X given Y is

$$\mathbf{Pr}\Big[X = x \,\Big|\, Y = y\Big] = \frac{\mathbf{Pr}\Big[(X = x) \cap (Y = y)\Big]}{\mathbf{Pr}\Big[Y = y\Big]}.$$

An equivalent and useful restatement of this is that

$$\mathbf{Pr}\Big[(X = x) \cap (Y = y)\Big] = \mathbf{Pr}\Big[X = x \;\Big|\; Y = y\Big] * \mathbf{Pr}\Big[Y = y\Big].$$

Definition 9.1.3. Two events X and Y are ***independent***, if $\mathbf{Pr}[X = x \cap Y = y] = \mathbf{Pr}[X = x] \cdot \mathbf{Pr}[Y = y]$. In particular, if X and Y are independent, then

$$\mathbf{Pr}\Big[X = x \;\Big|\; Y = y\Big] = \mathbf{Pr}[X = x].$$

Definition 9.1.4. The ***expectation*** of a random variable X is the average value of this random variable. Formally, if X has a finite (or countable) set of values, it is

$$\mathbf{E}\Big[X\Big] = \sum_{x} x \cdot \mathbf{Pr}[X = x],$$

where the summation goes over all the possible values of X.

One of the most powerful properties of expectations is that an expectation of a sum is the sum of expectations.

Lemma 9.1.5 (Linearity of expectation.). *For any two random variables X and Y, we have* $\mathbf{E}\left[X+Y\right] = \mathbf{E}\left[X\right] + \mathbf{E}\left[Y\right]$.

Proof: For the simplicity of exposition, assume that X and Y receive only integer values. We have that

$$\begin{aligned}
\mathbf{E}\left[X+Y\right] &= \sum_x \sum_y (x+y)\,\mathbf{Pr}\left[(X=x)\cap(Y=y)\right]\\
&= \sum_x \sum_y x * \mathbf{Pr}\left[(X=x)\cap(Y=y)\right] + \sum_x \sum_y y * \mathbf{Pr}\left[(X=x)\cap(Y=y)\right]\\
&= \sum_x x * \sum_y \mathbf{Pr}\left[(X=x)\cap(Y=y)\right] + \sum_y y * \sum_x \mathbf{Pr}\left[(X=x)\cap(Y=y)\right]\\
&= \sum_x x * \mathbf{Pr}\left[X=x\right] + \sum_y y * \mathbf{Pr}\left[Y=y\right]\\
&= \mathbf{E}\left[X\right] + \mathbf{E}\left[Y\right].
\end{aligned}$$

■

Another interesting creature, is the conditional expectation; that is, it is the expectation of a random variable given some additional information.

Definition 9.1.6. Given random variables X and Y, the ***conditional expectation*** of X given Y, is the quantity $\mathbf{E}\left[X \,\middle|\, Y\right]$. Specifically, you are given the value y of the random variable Y, and the $\mathbf{E}\left[X \,\middle|\, Y\right] = \mathbf{E}\left[X \,\middle|\, Y=y\right] = \sum_x x * \mathbf{Pr}\left[X=x \,\middle|\, Y=y\right]$.

Note, that for a random variable X, the expectation $\mathbf{E}[X]$ is a number. On the other hand, the conditional probability $f(y) = \mathbf{E}\left[X \,\middle|\, Y=y\right]$ is a function. The key insight why conditional probability is the following.

Lemma 9.1.7. *For any two random variables X and Y (not necessarily independent), we have that* $\mathbf{E}[X] = \mathbf{E}\left[\mathbf{E}\left[X \,\middle|\, Y\right]\right]$.

Proof: We use the definitions carefully:

$$\begin{aligned}
\mathbf{E}\left[\mathbf{E}\left[X \,\middle|\, Y\right]\right] &= \mathop{\mathbf{E}}_y\left[\mathbf{E}\left[X \,\middle|\, Y=y\right]\right] = \mathop{\mathbf{E}}_y\left[\sum_x x * \mathbf{Pr}\left[X=x \,\middle|\, Y=y\right]\right]\\
&= \sum_y \mathbf{Pr}\left[Y=y\right] * \left(\sum_x x * \mathbf{Pr}\left[X=x \,\middle|\, Y=y\right]\right)\\
&= \sum_y \mathbf{Pr}\left[Y=y\right] * \left(\sum_x x * \frac{\mathbf{Pr}\left[(X=x)\cap(Y=y)\right]}{\mathbf{Pr}[Y=y]}\right)\\
&= \sum_y \sum_x x * \mathbf{Pr}\left[(X=x)\cap(Y=y)\right] = \sum_x \sum_y x * \mathbf{Pr}\left[(X=x)\cap(Y=y)\right]\\
&= \sum_x x * \left(\sum_y \mathbf{Pr}\left[(X=x)\cap(Y=y)\right]\right) = \sum_x x * \mathbf{Pr}\left[X=x\right] = \mathbf{E}[X].
\end{aligned}$$

■

9.2 Sorting Nuts and Bolts

Problem 9.2.1 (Sorting Nuts and Bolts). You are given a set of n nuts and n bolts. Every nut have a matching bolt, and all the n pairs of nuts and bolts have different sizes. Unfortunately, you get the nuts and bolts separated from each other and you have to match the nuts to the bolts. Furthermore, given a nut and a bolt, all you can do is to try and match one bolt against a nut (i.e., you can not compare two nuts to each other, or two bolts to each other).

When comparing a nut to a bolt, either they match, or one is smaller than other (and you known the relationship after the comparison).

How to match the n nuts to the n bolts quickly? Namely, while performing a small number of comparisons.

The naive algorithm is of course to compare each nut to each bolt, and match them together. This would require a quadratic number of comparisons. Another option is to sort the nuts by size, and the bolts by size and then "merge" the two ordered sets, matching them by size. The only problem is that we can not sorts only the nuts, or only the bolts, since we can not compare them to each other. Indeed, we sort the two sets simultaneously, by simulating **QuickSort**. The resulting algorithm is depicted on the right.

```
MatchNutsAndBolts(N: nuts, B: bolts)
  Pick a random nut n_pivot from N
  Find its matching bolt b_pivot in B
  B_L ← All bolts in B smaller than n_pivot
  N_L ← All nuts in N smaller than b_pivot
  B_R ← All bolts in B larger than n_pivot
  N_R ← All nuts in N larger than b_pivot
  MatchNutsAndBolts(N_R,B_R)
  MatchNutsAndBolts(N_L,B_L)
```

9.2.1 Running time analysis

Definition 9.2.2. Let $\mathcal{RT}$ denote the random variable which is the running time of the algorithm. Note, that the running time is a random variable as it might be different between different executions on the *same input*.

Definition 9.2.3. For a randomized algorithm, we can speak about the expected running time. Namely, we are interested in bounding the quantity $\mathbf{E}[\mathcal{RT}]$ for the worst input.

Definition 9.2.4. The ***expected running-time*** of a randomized algorithm for input of size n is

$$T(n) = \max_{U \text{ is an input of size } n} \mathbf{E}\Big[\mathcal{RT}(U)\Big],$$

where $\mathcal{RT}(U)$ is the running time of the algorithm for the input U.

Definition 9.2.5. The ***rank*** of an element x in a set S, denoted by $\mathrm{rank}(x)$, is the number of elements in S of size smaller or equal to x. Namely, it is the location of x in the sorted list of the elements of S.

Theorem 9.2.6. *The expected running time of* **MatchNutsAndBolts** *(and thus also of* **QuickSort***) is $T(n) = O(n \log n)$, where n is the number of nuts and bolts. The worst case running time of this algorithm is $O(n^2)$.*

Proof: Clearly, we have that $\mathbf{Pr}[\text{rank}(n_{pivot}) = k] = \frac{1}{n}$. Furthermore, if the rank of the pivot is k then

$$
\begin{aligned}
T(n) &= \mathop{\mathbf{E}}_{k=\text{rank}(n_{pivot})}[O(n) + T(k-1) + T(n-k)] = O(n) + \mathop{\mathbf{E}}_{k}[T(k-1) + T(n-k)] \\
&= T(n) = O(n) + \sum_{k=1}^{n} \mathbf{Pr}[Rank(Pivot) = k] * (T(k-1) + T(n-k)) \\
&= O(n) + \sum_{k=1}^{n} \frac{1}{n} \cdot (T(k-1) + T(n-k)),
\end{aligned}
$$

by the definition of expectation. It is not easy to verify that the solution to the recurrence $T(n) = O(n) + \sum_{k=1}^{n} \frac{1}{n} \cdot (T(k-1) + T(n-k))$ is $O(n \log n)$. ■

9.2.1.1 Alternative incorrect solution

The algorithm **MatchNutsAndBolts** is lucky if $\frac{n}{4} \leq \text{rank}(n_{pivot}) \leq \frac{3}{4}n$. Thus, $\mathbf{Pr}[\text{“lucky”}] = 1/2$. Intuitively, for the algorithm to be fast, we want the split to be as balanced as possible. The less balanced the cut is, the worst the expected running time. As such, the “Worst” lucky position is when $\text{rank}(n_{pivot}) = n/4$ and we have that

$$T(n) \leq O(n) + \mathbf{Pr}[\text{“lucky”}] * (T(n/4) + T(3n/4)) + \mathbf{Pr}[\text{“unlucky”}] * T(n).$$

Namely, $T(n) = O(n) + \frac{1}{2} * \left(T(\frac{n}{4}) + T(\frac{3}{4}n)\right) + \frac{1}{2}T(n)$. Rewriting, we get the recurrence $T(n) = O(n) + T(n/4) + T((3/4)n)$, and its solution is $O(n \log n)$.

While this is a very intuitive and elegant solution that bounds the running time of **QuickSort**, it is also incomplete. The interested reader should try and make this argument complete. After completion the argument is as involved as the previous argument. Nevertheless, this argumentation gives a good back of the envelope analysis for randomized algorithms which can be applied in a lot of cases.

9.2.2 What are randomized algorithms?

Randomized algorithms are algorithms that use random numbers (retrieved usually from some unbiased source of randomness [say a library function that returns the result of a random coin flip]) to make decisions during the executions of the algorithm. The running time becomes a random variable. Analyzing the algorithm would now boil down to analyzing the behavior of the random variable $\mathcal{RT}(n)$, where n denotes the size of the input.In particular, the expected running time $\mathbf{E}[\mathcal{RT}(n)]$ is a quantity that we would be interested in.

It is useful to compare the expected running time of a randomized algorithm, which is

$$T(n) = \max_{U \text{ is an input of size } n} \mathbf{E}\Big[\mathcal{RT}(U)\Big],$$

to the worst case running time of a deterministic (i.e., not randomized) algorithm, which is

$$T(n) = \max_{U \text{ is an input of size } n} \mathcal{RT}(U),$$

```
FlipCoins
    while RandBit= 1 do
        nothing;
```

Caveat Emptor:[2]Note, that a randomized algorithm might have exponential running time in the worst case (or even unbounded) while having good expected running time. For example, consider the algorithm **FlipCoins** depicted on the right. The expected running time of **FlipCoins** is a geometric random variable with probability $1/2$, as such we have that $\mathbf{E}[\mathcal{RT}(\text{ FlipCoins })] = O(2)$. However, **FlipCoins** can run forever if it always gets 1 from the **RandBit** function.

This is of course a ludicrous argument. Indeed, the probability that **FlipCoins** runs for long decreases very quickly as the number of steps increases. It can happen that it runs for long, but it is extremely unlikely.

Definition 9.2.7. The running time of a randomized algorithm **Alg** is $O(f(n))$ with ***high probability*** if

$$\mathbf{Pr}\Big[\mathcal{RT}(\mathbf{Alg}(n)) \geq c \cdot f(n)\Big] = o(1).$$

Namely, the probability of the algorithm to take more than $O(f(n))$ time decreases to 0 as n goes to infinity. In our discussion, we would use the following (considerably more restrictive definition), that requires that

$$\mathbf{Pr}\Big[\mathcal{RT}(\mathbf{Alg}(n)) \geq c \cdot f(n)\Big] \leq \frac{1}{n^d},$$

where c and d are appropriate constants. For technical reasons, we also require that $\mathbf{E}[\mathcal{RT}(\mathbf{Alg}(n))] = O(f(n))$.

9.3 Analyzing QuickSort

The previous analysis works also for **QuickSort**. However, there is an alternative analysis which is also very interesting and elegant. Let $a_1, ..., a_n$ be the n given numbers (in sorted order – as they appear in the output).

It is enough to bound the number of comparisons performed by **QuickSort** to bound its running time, as can be easily verified. Observe, that two specific elements are compared to each other by **QuickSort** at most once, because **QuickSort** performs only comparisons against the pivot, and after the comparison happen, the pivot does not being passed to the two recursive subproblems.

Let X_{ij} be an indicator variable if **QuickSort** compared a_i to a_j in the current execution, and zero otherwise. The number of comparisons performed by **QuickSort** is **exactly** $Z = \sum_{i<j} X_{ij}$.

[2]Caveat Emptor - let the buyer beware (i.e., one buys at one's own risk)

Observation 9.3.1. *The element a_i is compared to a_j iff one of them is picked to be the pivot and they are still in the same subproblem.*

Also, we have that $\mu = \mathbf{E}[X_{ij}] = \mathbf{Pr}[X_{ij} = 1]$. To quantify this probability, observe that if the pivot is smaller than a_i or larger than a_j then the subproblem still contains the block of elements $a_i, \ldots, a_j$. Thus, we have that

$$\mu = \mathbf{Pr}[a_i \text{ or } a_j \text{ is first pivot} \in a_i, \ldots, a_j] = \frac{2}{j-i+1}.$$

Another (and hopefully more intuitive) explanation for the above phenomena is the following: Imagine, that before running **QuickSort** we choose for every element a random priority, which is a real number in the range $[0, 1]$. Now, we reimplement **QuickSort** such that it always pick the element with the lowest random priority (in the given subproblem) to be the pivot. One can verify that this variant and the standard implementation have the same running time. Now, a_i gets compares to a_j if and only if all the elements $a_{i+1}, \ldots, a_{j-1}$ have random priority larger than both the random priority of a_i and the random priority of a_j. But the probability that one of two elements would have the lowest random-priority out of $j - i + 1$ elements is $2 * 1/(j - i + 1)$, as claimed.

Thus, the running time of **QuickSort** is

$$\mathbf{E}\Big[\mathcal{RT}(n)\Big] = \mathbf{E}\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} \mathbf{E}\Big[X_{ij}\Big] = \sum_{i<j} \frac{2}{j-i+1} = 2\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{j-i+1}$$
$$= 2\sum_{i=1}^{n-1} \sum_{\Delta=2}^{n-i+1} \frac{1}{\Delta} \leq 2\sum_{i=1}^{n-1} \sum_{\Delta=1}^{n} \frac{1}{\Delta} \leq 2\sum_{i=1}^{n-1} H_n = 2nH_n.$$

by linearity of expectations, where $H_n = \sum_{i=1}^{n} \frac{1}{i} \leq \ln n + 1$ is the nth harmonic number,

As we will see in the near future, the running time of **QuickSort** is $O(n \log n)$ with high-probability. We need some more tools before we can show that.

9.4 QuickSelect – median selection in linear time

Consider the problem of given a set X of n numbers, and a parameter k, to output the kth smallest number (which is the number with ***rank*** k in X). This can be easily be done by modifying **QuickSort** only to perform one recursive call. See Figure 9.1 for a pseud-code of the resulting algorithm.

Intuitively, at each iteration of **QuickSelect** the input size shrinks by a constant factor, leading to a linear time algorithm.

Theorem 9.4.1. *Given a set X of n numbers, and any integer k, the expected running time of* **QuickSelect***(X, n) is $O(n)$.*

```
QuickSelect(X, k)
    // Input:  X = {x_1, ..., x_n} numbers, k.
    // Assume x_1, ..., x_n are all distinct.
    // Task:  Return kth smallest number in X.
    y ← random element of X.
    r ← rank of y in X.
    if r = k then return y
    X_< = all elements in X < than y
    X_> = all elements in X > than y
    // By assumption |X_<| + |X_>| + 1 = |X|.
    if r < k then
        return QuickSelect( X_>, k − r )
    else
        return QuickSelect( X_≤, k )
```

Figure 9.1: **QuickSelect** pseudo-code.

Proof: Let $X_1 = X$, and X_i be the set of numbers in the ith level of the recursion. Let y_i and r_i be the random element and its rank in X_i, respectively, in the ith iteration of the algorithm. Finally, let $n_i = |X_i|$. Observe that the probability that the pivot y_i is in the "middle" of its subproblem is

$$\alpha = \mathbf{Pr}\left[\frac{n_i}{4} \leq r_i \leq \frac{3}{4} n_i\right] \geq \frac{1}{2},$$

and if this happens then

$$n_{i+1} \leq \max(r_i - 1, n_i - r_i) \leq \frac{3}{4} n_i.$$

We conclude that

$$\begin{aligned}\mathbf{E}\left[n_{i+1} \,\middle|\, n_i\right] &\leq \mathbf{Pr}\left[y_i \text{ in the middle}\right] \frac{3}{4} n_i + \mathbf{Pr}\left[y_i \text{ not in the middle}\right] n_i \\ &\leq \alpha \frac{3}{4} n_i + (1-\alpha)\, n_i = n_i(1 - \alpha/4) \leq n_i(1 - (1/2)/4) = (7/8) n_i.\end{aligned}$$

Now, we have that

$$\begin{aligned}m_{i+1} &= \mathbf{E}\left[n_{i+1}\right] = \mathbf{E}\left[\mathbf{E}\left[n_{i+1} \,\middle|\, n_i\right]\right] \leq \mathbf{E}[(7/8)n_i] = (7/8)\,\mathbf{E}[n_i] = (7/8) m_i \\ &= (7/8)^i m_0 = (7/8)^i n,\end{aligned}$$

since for any two random variables we have that $\mathbf{E}[X] = \mathbf{E}\left[\mathbf{E}\left[X \,\middle|\, Y\right]\right]$. In particular, the expected running time of **QuickSelect** is proportional to

$$\mathbf{E}\left[\sum_i n_i\right] = \sum_i \mathbf{E}\left[n_i\right] \leq \sum_i m_i = \sum_i (7/8)^i n = O(n),$$

as desired. ∎