Chapter 1

Introduction, Quick Sort and BSP

By Sariel Har-Peled, January 19, 2018[1]

Everybody knows that the dice are loaded
Everybody rolls with their fingers crossed
Everybody knows the war is over
Everybody knows the good guys lost
Everybody knows the fight was fixed
The poor stay poor, the rich get rich
That's how it goes
Everybody knows

Everybody knows, Leonard Cohen

1.1. What are randomized algorithms?

Randomized algorithms are algorithms that makes random decision during their execution. Specifically, they are allowed to use variables, such that their value is taken from some random distribution. It is not immediately clear why adding the ability to use randomness helps an algorithm. But it turns out that the benefits are quite substantial. Before listing them, let start with an example.

1.1.1. The benefits of unpredictability

Consider the following game. The adversary has a equilateral triangle, with three coins on the vertices of the triangle (which are, numbered by, I don't known, 1,2,3). Initially, the adversary set each of the three coins to be either heads or tails, as she sees fit.

At each round of the game, the player can ask to flip certain coins (say, flip coins at vertex 1 and 3). If after the flips all three coins have the same side up, then the game stop. Otherwise, the adversary is allowed to rotate the board by $0, 120$ or -120 degrees, as she seems fit. And the game continues from this point on. To make things interesting, the player does not see the board at all, and does not know the initial configuration of the coins.

A randomized algorithm. The randomized algorithm in this case is easy – the player randomly chooses a number among $1, 2, 3$ at every round. Since, at every point in time, there are two coins that have the same side up, and the other coin is the other side up, a random choice hits the lonely coin, and thus finishes the game, with probability $1/3$ at each step. In particular, the number of iterations of the game till it terminates behaves like a geometric variable with geometric distribution with probability $1/3$ (and thus the expected number of rounds is 3). Clearly, the probability that the game continues for more than i rounds, when the player uses this random algorithm, is $(2/3)^i$. In particular, it vanishes to zero relatively quickly.

A deterministic algorithm. The surprise here is that there is no deterministic algorithm that can generate a winning sequence. Indeed, if the player uses a deterministic algorithm, then the adversary can simulate the algorithm herself, and know at every stage what coin the player would ask to flip (it is easy to verify that flipping two coins in a step is equivalent to flipping the other coin – so we can

restrict ourselves to a single coin flip at each step). In particular, the adversary can rotate the board in the end of the round, such that the player (in the next round) flips one of the two coins that are in the same state. Namely, the player never wins.

The shocker. One can play the same game with a board of size 4 (i.e., a square), where at each stage the player can flip one or two coins, and the adversary can rotate the board by $0, 90, 180, 270$ degrees after each round. Surprisingly, there is a deterministic winning strategy for this case. The interested reader can think what it is (this is one of these brain teasers that are not immediate, and might take you 15 minutes to solve, or longer [or much longer]).

The unfair game of the analysis of algorithms. The underlying problem with analyzing algorithm is the inherent unfairness of worst case analysis. We are given a problem, we propose an algorithm, then an all-powerful adversary chooses the worst input for our algorithm. Using randomness gives the player (i.e., the algorithm designer) some power to fight the adversary by being unpredictable.

1.1.2. Back to randomized algorithms

1. **Best.** There are cases where only randomized algorithms are known or possible, especially for games. For example, consider the 3 coins example given above.
2. **Speed.** In some cases randomized algorithms are considerably faster than any deterministic algorithm.
3. **Simplicity.** Even if a randomized algorithm is not faster, often it is considerably simpler than its deterministic counterpart.
4. **Derandomization.** Some deterministic algorithms arises from derandomizing the randomized algorithms, and this are the only algorithm we know for these problems (i.e., discrepancy).
5. **Adversary arguments and lower bounds.** The standard worst case analysis relies on the idea that the adversary can select the input on which the algorithm performs worst. Inherently, the adversary is more powerful than the algorithm, since the algorithm is completely predictable. By using a randomized algorithm, we can make the algorithm unpredictable and break the adversary lower bound.
   Namely, randomness makes the algorithm vs. adversary game a more balanced game, by giving the algorithm additional power against the adversary.

1.1.3. Randomized vs average-case analysis

Randomized algorithms are not the same as ***average-case analysis***. In average case analysis, one assumes that is given some distribution on the input, and one tries to analyze an algorithm execution on such an input.

On the other hand, randomized algorithms do not assume random inputs – inputs can be arbitrary. As such, randomized algorithm analysis is more widely applicable, and more general.

While there is a lot of average case analysis in the literature, the problem that it is hard to find distribution on inputs that are meaningful in comparison to real world inputs. In particular, for numerous cases, the average case analysis exposes structure that does not exist in real world input.

1.2. Basic probability

Here we recall some definitions about probability. The reader already familiar with these definition can happily skip this section.

1.2.1. Formal basic definitions: Sample space, σ-algebra, and probability

A ***sample space*** Ω is a set of all possible outcomes of an experiment. We also have a set of events $\mathcal{F}$, where every member of $\mathcal{F}$ is a subset of Ω. Formally, we require that $\mathcal{F}$ is a σ-algebra.

Definition 1.2.1. A single element of Ω is an ***elementary event*** or an ***atomic event***.

Definition 1.2.2. A set $\mathcal{F}$ of subsets of Ω is a ***σ-algebra*** if:

(i) $\mathcal{F}$ is not empty,
(ii) if $X \in \mathcal{F}$ then $\overline{X} = (\Omega \setminus X) \in \mathcal{F}$, and
(iii) if $X, Y \in \mathcal{F}$ then $X \cup Y \in \mathcal{F}$.

More generally, we require that if $X_i \in \mathcal{F}$, for $i \in \mathbb{Z}$, then $\cup_i X_i \in \mathcal{F}$. A member of $\mathcal{F}$ is an ***event***.

As a concrete example, if we are rolling a dice, then $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\mathcal{F}$ would be the power set of all possible subsets of Ω.

Definition 1.2.3. A ***probability measure*** is a mapping $\mathbf{Pr} : \mathcal{F} \to [0, 1]$ assigning ***probabilities*** to events. The function $\mathbf{Pr}$ needs to have the following properties:

(i) Additive: for $X, Y \in \mathcal{F}$ disjoint sets, we have that $\mathbf{Pr}\big[X \cup Y\big] = \mathbf{Pr}\big[X\big] + \mathbf{Pr}\big[Y\big]$, and
(ii) $\mathbf{Pr}[\Omega] = 1$.

Definition 1.2.4. A ***probability space*** is a triple $(\Omega, \mathcal{F}, \mathbf{Pr})$, where Ω is a sample space, $\mathcal{F}$ is a σ-algebra defined over Ω, and $\mathbf{Pr}$ is a probability measure.

Definition 1.2.5. A ***random variable*** f is a mapping from Ω into some set $\mathcal{G}$. We require that the probability of the random variable to take on any value in a given subset of values is well defined. Formally, for any subset $U \subseteq \mathcal{G}$, we have that $f^{-1}(U) \in \mathcal{F}$. That is, $\mathbf{Pr}[f \in U] = \mathbf{Pr}\big[f^{-1}(U)\big]$ is defined.

Going back to the dice example, the number on the top of the dice when we roll it is a random variable. Similarly, let X be one if the number rolled is larger than 3, and zero otherwise. Clearly X is a random variable.

We denote the ***probability*** of a random variable X to get the value x, by $\mathbf{Pr}[X = x]$ (or sometime $\mathbf{Pr}[x]$, if we are lazy).

1.2.2. Expectation and conditional probability

Definition 1.2.6 (Expectation). The expectation of a random variable X, is its average. Formally, the ***expectation*** of X is

$$\mathbf{E}\big[X\big] = \sum_x x \, \mathbf{Pr}\big[X = x\big] .$$

Definition 1.2.7 (Conditional Probability.). The ***conditional probability*** of X given Y, is the probability that $X = x$ given that $Y = y$. We denote this quantity by $\mathbf{Pr}[X = x \mid Y = y]$.

One useful way to think about the conditional probability $\mathbf{Pr}[X \mid Y]$ is as a function, between the given value of Y (i.e., y), and the probability of X (to be equal to x) in this case. Since in many cases x and y are omitted in the notation, it is somewhat confusing.

The conditional probability can be computed using the formula

$$\mathbf{Pr}[X = x \mid Y = y] = \frac{\mathbf{Pr}\big[(X = x) \cap (Y = y)\big]}{\mathbf{Pr}[Y = y]}.$$

For example, let us roll a dice and let X be the number we got. Let Y be the random variable that is true if the number we get is even. Then, we have that

$$\mathbf{Pr}\big[X = 2 \,\big|\, Y = \mathit{true}\big] = \frac{1}{3}.$$

Definition 1.2.8. Two random variables X and Y are ***independent*** if $\mathbf{Pr}\big[X = x \,\big|\, Y = y\big] = \mathbf{Pr}[X = x]$, for all x and y.

Observation 1.2.9. *If X and Y are independent then $\mathbf{Pr}\big[X = x \,\big|\, Y = y\big] = \mathbf{Pr}[X = x]$ which is equivalent to $\dfrac{\mathbf{Pr}[X = x \cap Y = y]}{\mathbf{Pr}[Y = y]} = \mathbf{Pr}[X = x]$. That is, X and Y are* independent*, if for all x and y, we have that*

$$\mathbf{Pr}\big[X = x \cap Y = y\big] = \mathbf{Pr}\big[X = x\big]\,\mathbf{Pr}\big[Y = y\big].$$

Remark. Informally, and not quite correctly, one possible way to think about conditional probability $\mathbf{Pr}[X = x \mid Y = y]$ is as measuring the benefit of having more information. If we know that $Y = y$, do we have any change in the probability of $X = x$?

Lemma 1.2.11 (Linearity of expectation). **Linearity of expectation** *is the property that for any two random variables X and Y, we have that $\mathbf{E}\big[X + Y\big] = \mathbf{E}\big[X\big] + \mathbf{E}\big[Y\big]$.*

Proof: $\mathbf{E}\big[X + Y\big] = \sum_{\omega \in \Omega} \mathbf{Pr}[\omega]\,\big(X(\omega) + Y(\omega)\big) = \sum_{\omega \in \Omega} \mathbf{Pr}[\omega]\,X(\omega) + \sum_{\omega \in \Omega} \mathbf{Pr}[\omega]\,Y(\omega) = \mathbf{E}\big[X\big] + \mathbf{E}\big[Y\big]$. ■

1.3. QuickSort

Let the input be a set $T = \{t_1, \ldots, t_n\}$ of n items to be sorted. We remind the reader, that the **QuickSort** algorithm randomly pick a pivot element (uniformly), splits the input into two subarrays of all the elements smaller than the pivot, and all the elements larger than the pivot, and then it recurses on these two subarrays (the pivot is not included in these two subproblems). Here we will show that the expected running time of **QuickSort** is $O(n \log n)$.

Definition 1.3.1. For an event $\mathcal{E}$, let X be a random variable which is 1 if $\mathcal{E}$ occurred and 0 otherwise. The random variable X is an ***indicator variable***.

Observation 1.3.2. *For an indicator variable X of an event $\mathcal{E}$, we have*

$$\mathbf{E}\Big[X\Big] = 0 \cdot \mathbf{Pr}\Big[X = 0\Big] + 1 \cdot \mathbf{Pr}\Big[X = 1\Big] = \mathbf{Pr}\Big[X = 1\Big] = \mathbf{Pr}\Big[\mathcal{E}\Big].$$

Let $S_1, \ldots, S_n$ be the elements in their sorted order (i.e., the output order). Let $X_{ij} = 1$ be the indicator variable which is one iff **QuickSort** compares S_i to S_j, and let p_{ij} denote the probability that this happens. Clearly, the number of comparisons performed by the algorithm is $C = \sum_{i<j} X_{ij}$. By linearity of expectations, we have

$$\mathbf{E}[C] = \mathbf{E}\Big[\sum_{i<j} X_{ij}\Big] = \sum_{i<j} \mathbf{E}[X_{ij}] = \sum_{i<j} p_{ij}.$$

We want to bound p_{ij}, the probability that the S_i is compared to S_j. Consider the last recursive call involving both S_i and S_j. Clearly, the pivot at this step must be one of $S_i, \ldots, S_j$, all equally likely. Indeed, S_i and S_j were separated in the next recursive call.

Observe, that S_i and S_j get compared if and only if pivot is S_i or S_j. Thus, the probability for that is $2/(j-i+1)$. Indeed,

$$p_{ij} = \mathbf{Pr}\big[S_i \text{ or } S_j \text{ picked} \,\big|\, \text{picked pivot from } S_i, \ldots, S_j\big] = \frac{2}{j-i+1}.$$

Thus,

$$\sum_{i=1}^{n}\sum_{j>i} p_{ij} = \sum_{i=1}^{n}\sum_{j>i} 2/(j-i+1) = \sum_{i=1}^{n}\sum_{k=1}^{n-i+1}\frac{2}{k} \le 2\sum_{i=1}^{n}\sum_{k=1}^{n}\frac{1}{k} \le 2nH_n \le n + 2n\ln n,$$

where H_n is the ***harmonic number***[2] $H_n = \sum_{i=1}^{n} 1/i$, We thus proved the following result.

Lemma 1.3.3. **QuickSort** *performs in expectation at most $n + 2n\ln n$ comparisons, when sorting n elements.*

Note, that this holds for all inputs. No assumption on the input is made. Similar bounds holds not only in expectation, but also with high probability.

This raises the question, of how does the algorithm pick a random element? We assume we have access to a random source that can get us number between 1 and n uniformly.

Note, that the algorithm always works, but it might take quadratic time in the worst case.

Remark 1.3.4 (Wait, wait, wait). Let us do the key argument in the above more slowly, and more carefully. 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.

1.4. Binary space partition (BSP)

Let assume that we would like to render an image of a three dimensional scene on the computer screen. The input is in general a collection of polygons in three dimensions. The *painter* algorithm, render the scene by drawing things from back to front; and let front stuff overwrite what was painted before.

[2] Using integration to bound summation, we have $H_n \le 1 + \int_{x=1}^{n} \frac{1}{x}dx \le 1 + \ln n$. Similarly, $H_n \ge \int_{x=1}^{n} \frac{1}{x}dx = \ln n$.

The problem is that it is not always possible to order the objects in three dimensions. This ordering might have cycles. So, one possible solution is to build a *binary space partition*. We build a binary tree. In the root, we place a polygon P. Let h be the plane containing P. Next, we partition the input polygons into two sets, depending on which side of h they fall into. We recursively construct a BSP for each set, and we hang it from the root node. If a polygon intersects h then we cut it into two polygons as split by h. We continue the construction recursively on the objects on one side of h, and the objects on the other side. What we get, is a binary tree that splits space into cells, and furthermore, one can use the painter algorithm on these objects. The natural question is how big is the resulting partition.

We will study the easiest case, of disjoint segments in the plane.

1.4.1. BSP for disjoint segments

Let $P = \{\mathsf{s}_1, \ldots, \mathsf{s}_n\}$ be n disjoint segments in the plane. We will build the BSP by using the lines defined by these segments. This kind of BSP is called ***autopartition***.

To recap, the BSP is a binary tree, at every internal node we store a segment of P, where the line associated with it splits its region into its two children. Finally, each leaf of the BSP stores a single segment. A ***fragment*** is just going to be a subsegment formed by this splitting. Clearly, every internal node, stores a fragment that defines its split. As such, the size of the BSP is proportional to the number of fragments generated when building the BSP.

One application of such a BSP is ray shooting - given a ray you would like to determine what is the first segment it hits. Start from the root, figure out which child contains the apex of the ray, and first (recursively) compute the first segment stored in this child that the ray intersect. Contain into the second child only if the first subtree does not contain any segment that intersect the ray.

1.4.1.1. The algorithm

We pick a random permutation σ of $1, \ldots, n$, and in the ith step we insert $\mathsf{s}_{\sigma(i)}$ splitting all the cells that s_i intersects.

Observe, that if s_i crosses a cell completely, it just splits it into two and no new fragments are created. As such, the bad case is when a segment s is being inserted, and its line intersect some other segment t.

So, let $\mathcal{E}(\mathsf{s},\mathsf{t})$ denote the event that when inserted s it had split t. In particular, let $\mathrm{index}(\mathsf{s},\mathsf{t})$ denote the number of segments on the line of s between s (closer) endpoint and t (including t. If the line of s does not intersect t, then $\mathrm{index}(\mathsf{s},\mathsf{t}) = \infty$.

We have that

$$\mathbf{Pr}\Big[\mathcal{E}(\mathsf{s},\mathsf{t})\Big] = \frac{1}{1+\mathrm{index}(\mathsf{s},\mathsf{t})}.$$

Let $X_{\mathsf{s},\mathsf{t}}$ be the indicator variable that is 1 if $\mathcal{E}(\mathsf{s},\mathsf{t})$ happens. We have that

$$S = \text{ number of fragments } = \sum_{i=1}^{n} \sum_{j=1, i\neq j}^{n} X_{\mathsf{s}_i,\mathsf{s}_j}.$$

As such, by linearity of expectations, we have

$$\begin{aligned}\mathbf{E}[S] &= \mathbf{E}\Big[\sum_{i=1}^{n}\sum_{j=1,i\neq j}^{n} X_{\mathsf{s}_i,\mathsf{s}_j}\Big] = \sum_{i=1}^{n}\sum_{j=1,i\neq j}^{n}\mathbf{E}\big[X_{\mathsf{s}_i,\mathsf{s}_j}\big] = \sum_{i=1}^{n}\sum_{j=1,i\neq j}^{n}\mathbf{Pr}\big[\mathcal{E}(\mathsf{s}_i,\mathsf{s}_j)\big]\\ &= \sum_{i=1}^{n}\sum_{j=1,i\neq j}^{n}\frac{1}{1+\operatorname{index}(\mathsf{s}_i,\mathsf{s}_j)}\\ &\leq \sum_{i=1}^{n}\sum_{j=1}^{n}\frac{2}{1+j} = 2nH_n.\end{aligned}$$

Since the size of the BSP is proportional to the number of fragments created, we have the following result.

Theorem 1.4.1. *Given n disjoint segments in the plane, one can build a BSP for them of size $O(n\log n)$.*

Csaba Tóth [Tót03] showed that BSP for segments in the plane, in the worst case, has complexity $\Omega\left(n\frac{\log n}{\log\log n}\right)$.

1.5. Extra: QuickSelect running time

We remind the reader that **QuickSelect** receives an array $t[1\ldots n]$ of n real numbers, and a number k, and returns the element of rank k in the sorted order of the elements of t. We can of course, use **QuickSort**, and just return the kth element in the sorted array, but a more efficient algorithm, would be to modify **QuickSelect**, so that it recurses on the subproblem that contains the element we are interested in. Formally, **QuickSelect** chooses a random pivot, splits the array according to the pivot. This implies that we now know the rank of the pivot, and if its equal to $\overline{\mathsf{m}}$, we return it. Otherwise, we recurse on the subproblem containing the required element (modifying $\overline{\mathsf{m}}$ as we go down the recursion. Namely, **QuickSelect** is a modification of **QuickSort** performing only a single recursive call (instead of two).

As before, to bound the expected running time, we will bound the expected number of comparisons. As before, let $S_1,\ldots,S_n$ be the elements of t in their sorted order. Now, for $i<j$, let X_{ij} be the indicator variable that is one if S_i is being compared to S_j during the execution of **QuickSelect**. There are several possibilities to consider:

(i) If $i<j<\overline{\mathsf{m}}$: Here, S_i is being compared to S_j, if and only if the first pivot in the range $S_i,\ldots,S_k$ is either S_i or S_j. The probability for that is $2/(k-i+1)$. As such, we have that

$$\alpha_1 = \mathbf{E}\left[\sum_{i<j<\overline{\mathsf{m}}} X_{ij}\right] = \mathbf{E}\left[\sum_{i=1}^{\overline{\mathsf{m}}-2}\sum_{j=i+1}^{\overline{\mathsf{m}}-1} X_{ij}\right] = \sum_{i=1}^{\overline{\mathsf{m}}-2}\sum_{j=i+1}^{\overline{\mathsf{m}}-1}\frac{2}{\overline{\mathsf{m}}-i+1} = \sum_{i=1}^{med-2}\frac{2(\overline{\mathsf{m}}-i-1)}{\overline{\mathsf{m}}-i+1} \leq 2\Big(\overline{\mathsf{m}}-2\Big).$$

(ii) If $\overline{\mathsf{m}}<i<j$: Using the same analysis as above, we have that $\mathbf{Pr}\big[X_{ij}=1\big] = 2/(j-\overline{\mathsf{m}}+1)$. As such,

$$\alpha_2 = \mathbf{E}\left[\sum_{j=\overline{\mathsf{m}}+1}^{n}\sum_{i=\overline{\mathsf{m}}+1}^{j-1} X_{ij}\right] = \sum_{j=\overline{\mathsf{m}}+1}^{n}\sum_{i=\overline{\mathsf{m}}+1}^{j-1}\frac{2}{j-\overline{\mathsf{m}}+1} = \sum_{j=\overline{\mathsf{m}}+1}^{n}\frac{2(j-\overline{\mathsf{m}}-1)}{j-\overline{\mathsf{m}}+1} \leq 2\Big(n-\overline{\mathsf{m}}\Big).$$

(iii) $i < \overline{\mathsf{m}} < j$: Here, we compare S_i to S_j if and only if the first indicator in the range $S_i, \ldots, S_j$ is either S_i or S_j. As such, $\mathbf{E}\left[X_{ij}\right] = \mathbf{Pr}\left[X_{ij} = 1\right] = 2/(j-i+1)$. As such, we have

$$\alpha_3 = \mathbf{E}\left[\sum_{i=1}^{\overline{\mathsf{m}}-1} \sum_{j=\overline{\mathsf{m}}+1}^{n} X_{ij}\right] = \sum_{i=1}^{\overline{\mathsf{m}}-1} \sum_{j=\overline{\mathsf{m}}+1}^{n} \frac{2}{j-i+1}.$$

Observe, that for a fixed $\Delta = j - i + 1$, we are going to handle the gap Δ in the above summation, at most $\Delta - 2$ times. As such, $\alpha_3 \leq \sum_{\Delta=3}^{n} 2(\Delta-2)/\Delta \leq 2n$.

(iv) $i = \overline{\mathsf{m}}$. We have $\alpha_4 = \displaystyle\sum_{j=\overline{\mathsf{m}}+1}^{n} \mathbf{E}\left[X_{ij}\right] = \sum_{j=\overline{\mathsf{m}}+1}^{n} \frac{2}{j-\overline{\mathsf{m}}+1} = \ln n + 1.$

(v) $j = \overline{\mathsf{m}}$. We have $\alpha_5 = \displaystyle\sum_{i=1}^{\overline{\mathsf{m}}-1} \mathbf{E}\left[X_{ij}\right] = \sum_{i=1}^{\overline{\mathsf{m}}-1} \frac{2}{\overline{\mathsf{m}}-i+1} \leq \ln \overline{\mathsf{m}} + 1.$

Thus, the expected number of comparisons performed by **QuickSelect** is bounded by

$$\sum_i \alpha_i \leq 2(\overline{\mathsf{m}} - 2) + 2(n - \overline{\mathsf{m}}) + 2n + \ln n + 1 + \ln \overline{\mathsf{m}} = 4n - 2 + \ln n + \ln \overline{\mathsf{m}}.$$

Theorem 1.5.1. *In expectation,* **QuickSelect** *performs at most* $4n - 2 + \ln n + \ln \overline{\mathsf{m}}$ *comparisons, when selecting the* $\overline{\mathsf{m}}$*th element out of* n *elements.*

A different approach can reduce the number of comparisons (in expectation) to $1.5n + o(n)$. More on that later in the course.

Bibliography

[Tót03] C. D. Tóth. A note on binary plane partitions. *Discrete Comput. Geom.*, 30(1):3–16, 2003.