Randomized Searching and Sorting

Kent Quanrud

April 15, 2021

1 Introduction

Probability theory is simple and intuitive. Let us start with a casual example.

Suppose we flip a coin in the air, and press pause. Will it land heads or tails? Obviously, *we don't know yet*. But we can state without ambiguity that *half the time it will land heads*, and *half the time it will land tails*. What does it mean to say that half the time it will land heads? There is of course only one coin, and we can't split the coin in half. We are imagining that, if we repeat the experiment many times, we would expect half the coin tosses to come up heads.

This simple example, which we all understand throughouly, points to a deeper feature of probability: *probability allows us to interpret fractional values as discrete ones*. Here, "half heads" does not mean that "half the coin will come up heads", which is total nonsense; rather it means that half *the time* the coin will come up heads.

The formal rules of probability are simple. (The only tricky part is sticking to them!) We assume the reader has some acquaintance already with random events and variables, but we will still review the basics. Probability theory assumes an uncertain world where *events* occur with fixed probabilities. Each event A had a probability between 0 and 1, denoted

$$\mathbf{P}[A] \in [0, 1].$$

For every event A, there is the complementary event, $\bar{A}$, of A *not occurring*. We always have

$$\mathbf{P}[A] + \mathbf{P}[\bar{A}] = 1.$$

For any two events A and B, one can define the event that **both A and B occur**: denoted

$$\text{“}A \wedge B\text{”} \text{ or “}A \cap B\text{”} \text{ or “}A \text{ and } B\text{”}.$$

We also have the event that **either A or B** occurs, denoted

$$\text{“}A \vee B\text{”} \text{ or “}A \cup B\text{”} \text{ or “}A \text{ or } B\text{”}.$$

Their probabilities satisfy

$$\mathbf{P}[A \wedge B] \leq \min\{\mathbf{P}[A], \mathbf{P}[B]\}.$$

and

$$\mathbf{P}[A \vee B] \leq \mathbf{P}[A] + \mathbf{P}[B].$$

The second inequality, called the **union bound**, is surprisingly useful. One can also consider intersections and unions of more than two variables, which lead to the same inequalities as above.

A *finite random variables* where a random variable X takes one of a finite set of values, $\{x_1, \ldots, x_k\}$.[1] For each outcome x_i, "X equals x_i" is an event, with a fixed probability, denoted $\mathbf{P}[X = x_i]$. These probabilities sum to 1:

$$\sum_{i=1}^{k} \mathbf{P}[X = x_i] = 1.$$

For example, we can describe a coin toss as a random variable $X \in \{\texttt{heads}, \texttt{tails}\}$. Let a fair coin be tossed. If the coin comes up heads, then $X = \texttt{heads}$. If the coin comes up tails, then $X = \texttt{tails}$. We have

$$\mathbf{P}[X = \texttt{heads}] = \mathbf{P}[X = \texttt{tails}] = \frac{1}{2}.$$

Observe that these probabilities sum to 1.

If we have two random variables $X \in \{x_1, \ldots, x_k\}$ and $Y \in \{y_1, \ldots, y_\ell\}$, then their product (X, Y) forms a random variable in the set $\{(x_i, y_j) : i = 1, \ldots, k,\ j = 1, \ldots, \ell\}$. We have probabilities of the form

$$\mathbf{P}\big[X = x_i,\ Y = Y_j\big]$$

that gives the probability that $(X, Y) = (x_i, y_j)$. It is possible, but *not necessarily* the case, that

$$\mathbf{P}[X = x_i,\ Y = y_k] = \mathbf{P}[X = x_i]\,\mathbf{P}[Y = y_k]$$

In the special case where the above holds for all x_i and y_j, then X and Y are **independent**.

For example, suppose $X, Y \in \{\texttt{heads}, \texttt{tails}\}$ describe coin tosses. If they described different coin tosses, then they would be independent random variables, and each combination of heads and tails would occur with probability .25. That is,

$$\begin{aligned}\mathbf{P}[X = \texttt{heads},\ Y = \texttt{heads}] &= \mathbf{P}[X = \texttt{heads},\ Y = \texttt{tails}] \\ = \mathbf{P}[X = \texttt{tails},\ Y = \texttt{heads}] &= \mathbf{P}[X = \texttt{tails},\ Y = \texttt{tails}] = \frac{1}{4},\end{aligned}$$

Thus X and Y are independent random variables. If they described the *same coin*, then we would have

$$\mathbf{P}[X = \texttt{heads},\ Y = \texttt{heads}] = \mathbf{P}[X = \texttt{tails},\ Y = \texttt{tails}] = \frac{1}{2},$$

while

$$\mathbf{P}[X = \texttt{heads},\ Y = \texttt{tails}] = \mathbf{P}[X = \texttt{tails},\ Y = \texttt{heads}] = 0.$$

Here, X and Y are *not* independent.

[1] One can also define continuous variable (e.g., that take values continuously between 0 and 1), where sums are replaced by variables.

1.1 Expectations and linearity

When a random variable X takes on real values, we can have a well-defined and quantitative notion of "averages", called the *expected value*.

Definition 1. *Let $X \in \mathbb{R}$ be a real-valued random variable that has a finite set of possible values. Then the* **expected value of X**, *denoted* $\mathbf{E}[X]$, *is the weighted sum*

$$\mathbf{E}[X] \stackrel{\text{def}}{=} \sum_x \mathbf{P}[X = x] \cdot x;$$

where the sum is over all values of x where $\mathbf{P}[X = x] > 0$.

For continuous random variables, the sum would be replaced by an integral.

The average quantity of a random variable is very intuitive.

Theorem 2 (Linearity of expectation)**.** *Let $X, Y \in \mathbb{R}$ be two random variables. Then*

$$\mathbf{E}[X + Y] = \mathbf{E}[X] + \mathbf{E}[Y].$$

The proof of linearity of expectation is left as Exercise 1. The reader may want to first consider the simple case where $X \in \{x_1, x_2\}$ takes on exactly two values, and $Y \in \{y_1, y_2\}$ takes on exactly two values. One can generalize to generally finite sets from there.

Observe that linearity of expectation does not make any assumptions about how X and Y are structured or related. This makes linearity of expectation extremely useful and often leads to surprising observations.

A simple example of linearity of expectation is as follows. Consider a population of people with various heights. Let X and Y be two quantities obtained by the following experiment. Draw one person uniformly at random. Let X be the length from the waist of this person to the top of their head. Let Y be the length from the waist of this person to the ground. $X + Y$ gives to the total height of the person. Note that X and Y are highly dependent, since they both measure the same (randomly drawn) person. Linearity of expectation says:

$$\underbrace{\left(\text{average height}\right)}_{\mathbf{E}[X+Y]} = \underbrace{\begin{pmatrix}\text{average length}\\ \text{from waist up}\end{pmatrix}}_{\mathbf{E}[X]} + \underbrace{\begin{pmatrix}\text{average length}\\ \text{from waist down}\end{pmatrix}}_{\mathbf{E}[Y]}.$$

Of course, this makes total sense.

2 Randomized sorting

Let us start with the very basics: sorting. Recall that the goal is to take an unordered list of comparable elements (e.g., numbers) and return them in a list in sorted order. Previously we studied the the `merge-sort` algorithm which ran in $O(n \log n)$. Here we will study a randomized algorithm that is remarkably simple, called `quick-sort`, that is often the preferred one is very simple. The ideas every simple: choose a pivot *uniformly at random*, divide the elements by the pivot, and recurse on both halves.

`quick-sort(`$A[1..n]$`)`

// For simplicity we assume all the elements are distinct. Otherwise, break ties consistently.

1. `If` $n \leq 1$ `then return` A
2. `Select` $i \in [n]$ `uniformly at random.`
3. $B[1..k] \leftarrow$ `recursively sort the set of elements less than` $A[i]$
4. $C[1..\ell] \leftarrow$ `recursively sort the set of elements greater than` $A[i]$
5. `Return the concatenation of` B, $A[i]$, `and` C.

The running time is proportional to the total number of comparisons made by the algorithm. It is not impossible that the algorithm makes $O(n^2)$ comparisons (how?). However, the algorithm is randomized, and a more useful measure is the *average* number of comparisons.

We remark that a probabilistic analysis of `quick-sort` seems much more complicated than typical exercises in probability. We are analyzing a randomized *algorithm*, where one randomized decision affects the dynamics for all randomized decision. There are overwhelmingly many "butterfly effects" to consider.

The saving grace is that we are not trying to map out all the probabilistic outcomes with complete precision. We will only be interested in analyzing the running time on *average*; that is, in the *aggregate* in a certain probabilistic sense. We will show that `quick-sort` takes $O(n \log n)$ time in expectation *against any input*. This is still a *worst case analysis* in the sense that it holds for *any input*. This is not to be confused with the performance of an algorithm against a *randomized input from a particular distribution* – that is called *average case analysis*.

Theorem 3. *Given a list of n comparable elements,* `quick-sort` *returns elements in a sorted list in* $O(n \log n)$ *expected time.*

Proof. For each $i, j \in [n]$ with $i < j$, let X_{ij} be equal to 1 if the rank i element (i.e., the ith smallest element) is compared to the rank j element, and 0 otherwise. $\sum_{i<j} X_{ij}$ represents the total number of comparisons made by the algorithm. Let us first consider X_{ij} for fixed $i < j$. Observe that the rank i and rank j numbers are compared to each other iff either is selected as the pivot before any element of rank between i and j. Since the pivots are selected uniformly at random, this occurs with probability $2/(j - i + 1)$. That is,

$$\mathbf{E}\big[X_{ij}\big] \overset{\text{(a)}}{=} \mathbf{P}\big[X_{ij} = 1\big] = \frac{2}{j - i + 1}.$$

Note that (a) holds for any $\{0, 1\}$-random variable.

Let us now return to the entire sum, $\sum_{i<j} X_{ij}$. While each X_{ij} was simple to analyze alone, the different X_{ij}'s are not at all independent. How do we analyze them *en masse*. Enter *linearity of expectation*. We have

$$\mathbf{E}\left[\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{ij}\right] \overset{\text{(b)}}{=} \sum_{i=1}^{n}\sum_{j=i+1}^{n} \mathbf{E}\big[X_{ij}\big] \overset{\text{(c)}}{=} \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n}\sum_{k=1}^{n-i} \frac{2}{k} \leq O(n \log n).$$

Here (b) applies linearity of expectation: the average sum is equal to the sum of averages. (c) is from our analysis for a single X_{ij} above. ∎

Remark 2.1. Moreover, one can show that the running time is $O(n \log n)$ with exceedingly high probability – though to prove this we need generalizations of the law of large numbers called *concentration inequalities*.

3 Randomized approximations for SAT

Recall that SAT problem. We have a Boolean formula $f(x_1, \ldots, x_n)$ in conjunctive normal form, with n variables and m clauses. The goal is to find an assignment $x_1, \ldots, x_n \in \{\texttt{true}, \texttt{false}\}$ that satisfies as many classes as possible.

As we know, we do not know how to solve this exactly in polynomial time and it is considered unlikely that we will. So instead researchers often develop *approximation algorithms* for SAT. Here the goal is to find an assignment that satisfies as many clauses as possible. Of course an exact algorithm for this maximization version implies a polynomial time algorithm. Instead we will design algorithms that are *provably* competitive with the optimum solution.

Given a SAT formula f, let OPT denote the maximum number of clauses that are satisfiable. For $\alpha \in [0, 1]$, an **α-approximation algorithm** for SAT is an algorithm that produces an assignment that satisfies at least α OPT clauses. While obtaining an exact algorithm is NP-Hard, for $\alpha < 1$, it is not necessarily NP-Hard to obtain an α-approximation algorithm for SAT. Here we will develop *randomized* approximation algorithms that obtain

Consider the special case of 3-SAT. Here we have m clauses, each of which have exactly 3 variables.

`random-SAT(`$f(x_1, \ldots, x_n)$`)`

1. `for each` $i \in [n]$, `draw` $x_i \in \{\texttt{true}, \texttt{false}\}$ `independently and uniformly at random.`
2. `return` $x_1, \ldots, x_n$

Consider a single clause; e.g., $(x_1 \wedge \bar{x}_2 \wedge x_3)$. Of all ways to assign the three variables (x_1, x_2, x_3 in the example) values in $\{\texttt{true}, \texttt{false}\}$, there is only one way that does *not* satisfy the clause. That is,

each clause is satisfied with probability $7/8$.

In expectation, we have

$$\mathbf{E}[\text{clauses satisfied}] \overset{\text{(d)}}{=} \sum_{i=1}^{m} \mathbf{P}[i\text{th clause is satisfied}] = \frac{7}{8}m.$$

(d) applies linearly of expectation. Since $m \geq \text{OPT}$, we obtain a 7/8-approximation ratio (in expectation).

It is easy to extend the above to k-SAT for any $k \in \mathbb{N}$, where each clause has exactly k variables, and obtain the following.

Theorem 4. *For all $k \in \mathbb{N}$, there is a randomized $(1 - 1/2^k)$-approximation algorithm for k-SAT.*

The above algorithm is so simple, and essentially oblivious to the input f, that it is hard to believe it is a very good algorithm. Remarkably, there is reason to believe that it is the best possible polynomial time algorithm unless $P = NP$. The **PCP theorem** states that for all constants $\epsilon > 0$, getting better than a $(7/8 + \epsilon)$-approximation to 3SAT is NP-Hard. The PCP theorem gives similar *hardness of approximation* results for many other problems besides SAT, and in general, has wide and deep consequences across theoretical computer science. The PCP theorem is far beyond the scope of this class although we did cover the proof in the Fall 2020 randomized algorithms course [3].

4 Randomized minimum cut

Recall the *minimum cut* problem in undirected graphs. The input consists of a connected, undirected graph $G = (V, E)$ with positive edge capacities $c : E \to \mathbb{R}_{>0}$. A **cut** is a set of edges $C \subseteq E$ whose removal disconnects the graph. The goal is to

$$\text{minimize } \sum_{e \in C} c(e) \text{ over all cuts } C \subset E. \tag{1}$$

This problem is polynomial time solvable. Whatever the optimum cut is, it must be a minimum (s, t)-cut for some pair of vertices s and t. Thus to find the global minimum, one can guess s and t by looping over V, and compute the minimum $\{s, t\}$-cut for each choice of s and t. (Better yet: fix s, and loop over all t.) Previously, we also studied an algorithm due to [2] that runs in $O(mn)$ time.

For a set of vertices S, let $\partial(S)$ denote the set of edges with exactly one endpoint in S. $\partial(S)$ is called the cut **induced by** S. The induced cuts are also the inclusionwise minimal cuts, and it suffices to consider only the induced cuts when solving (1).

We will study a stunning algorithm discovered by Karger [1], that has had many implications beyond minimum cut. Consider the following description of Karger's algorithm.

> *Repeatedly sample edges in proportion to their capacities until there is only one cut from which we have not yet sampled any edges. Return this cut.*

This algorithm is clearly ridiculous. For the unweighted setting, the above algorithm is equivalent to the following, equally absurd approach (see Exercise 3).

> *Independently assign every edge $e \in E$ a weight $w_e \in [0, 1]$ uniformly at random. Build the minimum weight spanning tree T w/r/t w. Let e be the heaviest edge in T. Return the cut induced by the two components of $T - e$.*

Compare the two approaches above. Of course we know how to compute the minimum spanning tree; among other approaches, we can repeatedly add the smallest weight edge to T that does not create a cycle. On the other hand, in the first approach, it might appear difficult to keep track of which cuts we have and have not sampled from, being that there are so many cuts. This can be addressed by *contracting the graph*. Suppose we sample an edge $e = \{s, t\}$. Then we know that any cut $\partial(S)$, where $s \in S$ and $t \notin S$, has now been

```
random-contractions(G = (V, E), c)

1. while |E| > 1

   A. sample e ~ c

   B. G ← G/e,  c ← c/e

2. let E = {e}

3. return edges in the original graph that contracted to e
```

Figure 1: A randomized minimum cut algorithm due to Karger [1].

sampled from. Thus we can safely **contract** e; replacing s and t with a single vertex u that has the sum[2] of edges incident to s and t. Note that contracting e will only effect cuts that contain e. Now imagine we contract edges as we sample them. Eventually there are only two vertices left in the contracted graph, which represent two connected components in the input graph. These components induce the only cut we have not yet sampled from, and this is the cut that we return.

For an edge $e \in G$, we let $G/e = (V/e, E/e)$ denote the graph obtained by contracting e, and we let $c/e : E/e \to \mathbb{R}_{>0}$ denote the corresponding capacities. Pseudocode for the contraction algorithm is given in Figure 1.

The intuition behind Figure 1 is as follows. Here we describe the intuition for unweighted graphs for the sake of concreteness. (The intuition is the same for weighted graphs, except replacing "many edges" with "large capacity", etc.) Suppose we have an unweighted graph $G = (V, E)$, and let $C \subset E$ be the minimum cut. Since C is the minimum cut – keyword minimum – there are presumably very few edges in C. If we randomly sample an edge $e \in E$, then hopefully $e \notin C$. If C "survives" this round, then we have all made some progress because there is one less vertex in the graph after contracting e. In the next round, C is still the minimum cut, so the high-level logic from the first round still holds. Thus we can repeatedly sample edges and preserve the hope that we avoid C.

The above argument hinges on *how much smaller* C is then E. If we can argue that C is always a miniscule fraction of E, then we might hope that C survives to the end, after all. On the other hand, if C is even a small constant fraction of G, we will probably sample from C after a constant number of rounds. Observe also that over time, C becomes a larger and larger fraction of E, as we contract and remove edges outside of C.

The key observation is that *every vertex v induces a cut $\partial(v)$, which must have at least as many edges as C. Thus the minimum cut is at most the minimum degree in the graph.* In turn, since the number of edges in E is the sum of degrees (divided by 2), *the minimum cut C is at*

[2]More precisely, for every edge f of the form $\{s, z\}$ or $\{t, z\}$, we create a new edge $\{u, z\}$ with the same capacity. If s and t both have edges to the same vertex z, we can either create two edges from u to z with the appropriate capacities, or make a single edge from u to v with the same capacity. We remove s and t and its incident edges from the graph, replacing them with u and the newly created edges incident to u.

most a $2/n$ fraction of the total number of edges! This observation holds initially in the input graph and thereafter in the contracted graphs, although n decreases by 1 in each iteration.

On the first iteration, C has at most a $2/n$ chance of being hit. On the second interation, assuming C surivived the first iteration, C has (at most) a $2/(n-1)$ chance of being hit. Continuing in this fashion, assuming C surivived the first $i-1$ iterations, C has a $2/(n-i+1)$ change of being hit in the ith iteration. If one combines these problems, one discovers that C has a $\geq 1/\binom{n}{2}$ chance of surviving all $n-1$ rounds We can repeat the experiment $\binom{n}{2} = O(n^2)$ (a polynomial!) number of times to find the minimum cut with constant probability, and $O(n^2 \log n)$ times to find the minimum cut with high probability.

In the sequel, we formalize the the above argument, as well as extend it to positive capacities. For ease of notation, for a set of edges $C \subset E$, we denote the sum of capacities over C by

$$\bar{c}(C) \stackrel{\text{def}}{=} \sum_{e \in C} c(e).$$

Lemma 4.1. *Let $C^\star$ be the minimum cut in (G, c), and suppose $e \notin C$. Then $C^\star$ is (or maps to) to the minimum cut in the contracted graph $(G/e, c/e)$.*

Proof sketch. Direct inspection. ■

Lemma 4.2. $\bar{c}(E) \geq \frac{\lambda n}{2}$.

Proof. Every vertex v has weighted degree $\bar{c}(\partial(v)) \geq \lambda$ since $\partial(v)$ is a cut. Thus

$$\bar{c}(E) = \frac{\sum_v \bar{c}(\partial(v))}{2} \geq \frac{\lambda n}{2}.$$

■

Lemma 4.3. *Let $e \sim c$. Then $\mathbf{P}[e \in C^\star] \leq \frac{2}{n}$.*

Proof. We have $\mathbf{P}[e \in C^\star] = \frac{\bar{c}(C^\star)}{\bar{c}(E)} \stackrel{\text{(a)}}{\leq} \frac{2}{n}$ by (a)Lemma 5.2. ■

Lemma 4.4. *Let $C^\star$ be a minimum cut. With probability $\geq 1/\binom{n}{2}$,* `random-contractions` *returns $C^\star$.*

Proof. For $k \in \mathbb{Z}_{\geq 0}$, let E_k be the event that we have not sampled $C^\star$ after k iterations. Initially, $\mathbf{P}[E_0] = 1$, and we want to show that $\mathbf{P}[E_{n-2}] \geq 1/\binom{n}{2}$. By Lemma 5.3, we have

$$\mathbf{P}[E_k \mid E_{k-1}] \geq 1 - \tfrac{2}{n-(k-1)} \text{ for each } k \in [n].$$

The probability of succeeding (event E_{n-2}) is at least

$$\begin{aligned}\mathbf{P}[E_{n-2}] &= \prod_{k=1}^{n-2} \mathbf{P}[E_k \mid E_{k-1}] \geq \prod_{i=3}^{n} \left(1 - \frac{2}{i}\right) \\ &= \prod_{i=3}^{n} \frac{i-2}{i} = \frac{(n-2)!2}{n!} = \frac{1}{\binom{n}{2}}.\end{aligned}$$

■

Thus with probability about $1/n^2$, the random contraction algorithm returns the minimum cut. To find the minimum cut with constant probability, we rerun the algorithm $O(n^2)$ time and return the best cut. To find the minimum cut with *high probability*, we rerun the algorithm $O(n^2 \log n)$ times.

The nice thing about repetition is that we can run the randomized trials *in parallel*. Moreover, a single instance of the contraction algorithm (via its connection to minimum spanning trees) can be made to run in polylog(n) time with polynomially many processors. Thus one obtains a randomized parallel algorithm for minimum cut.

Corollary 5. *A randomized minimum cut can be computed in parallel in polylogarithmic time with a polynomial number of processors.*

That said, `random-contractions` is not just an algorithm. It is also a profound structural observation about the number of minimum cuts in an undirected graph. In the above algorithm, any fixed minimum cut is returned with probability $1/\binom{n}{2}$. This implies that there are at most $\binom{n}{2}$ minimum cuts in the graph!

Corollary 6. *There are at most $\binom{n}{2}$ minimum cuts in a graph.*

5 Exercises

Exercise 1. Prove linearity of expectation (Theorem 2).

Exercise 2. Recall the selection problem. Given an input $A[1..n]$ of n comparable elements, the goal is to find the rank k element (that is, the kth largest element) in A. You may assume the elements are distinct for simplicity.

Previously we studied a deterministic, linear time algorithm called **median-of-medians**. The algorithm was rather non-trivial – it recursively computes an approximate median to be use as a pivot. Here we wills study a much simpler algorithm called `quick-select` that uses randomization instead of recursion. The idea is very simple and similar to `quick-sort`: *select a pivot randomly*. In practice, this simpler algorithm is faster than `median-of-medians`.

`quick-select(`$A[1..n], k$`)`

// *The goal is to find the rank k element in $A[1..n]$. We assume for simplicity that all the elements are distinct.*

1. Randomly select $i \in [n]$ uniformly at random.
2. Compute the rank ℓ of $A[i]$
3. Cases:
   - A. $\ell = k$: return $A[i]$.
   - B. $\ell > k$: recursively search for the rank k element among the set of $\ell - 1$ elements less than $A[\ell]$ and return it.
   - C. $\ell < k$: recursively search for the rank $k - \ell$ element among the set of $n - \ell$ elements greater than $A[\ell]$ and return it.

The goal of this exercise to prove that `quick-select` takes $O(n)$ time in expectation. We will prove it in two different ways which offer two different perspectives (we expect you to do both).

1. **Approach 1.** Analyze `quick-select` similarly to `quick-sort`, based on the sum of indicators X_{ij}.

   You may want to do a separate analysis for each of the following 4 classes of pairs:

   (a) X_{ij} where $i < j < k$,

   (b) X_{ij} where $i < k < j$,

   (c) X_{ij} where $k < i < j$, and

   (d) X_{ij} where either $i = k$ or $j = k$.

   For each class, show that the expected sum is $O(n)$.[3] Use this to obtain a $O(n)$ expected running time, overall.

2. **Approach 2.** The following approach can be interpreted as a randomized divide and conquer argument. We are arguing that with constant probability, we decrease the input by a constant factor, from which the fast (expected) running time follows.

   (a) Consider a coin that flips heads with probability p. Suppose we repeatedly flip the coin until it comes up heads. What is the expected number of coin tosses until it comes up heads? Prove your answer.[4]

   (b) Consider again `quick-select`. Consider a single iteration where we pick a pivot uniformly at random and throw out some elements. Prove that with some constant probability p, we throw out at least $1/4$ of the remaining elements.

   (c) For each integer i, prove that the expected number of iterations (i.e., rounds of choosing a pivot) of `quick-select`, where the number of elements remaining is in the range $[(4/3)^i, (4/3)^{i+1})$, is $O(1)$.

   (d) Fix an integer i, and consider the amount of time spent by `quick-select` while the number of elements remaining is greater than $(4/3)^{i-1}$ and at most $(4/3)^i$. Show that that the expected amount of time is $\leq O\big((4/3)^i\big)$

   (e) Finally, use the preceding part to show that the expected running time of `quick-select` is $O(n)$.

Exercise 3. Consider the randomized algorithm for minimum cut based on building the minimum spanning tree w/r/t randomized weights, described in Section 5.

1. Prove that this algorithm is equivalent to the random contractions algorithm for unweighted graphs.

[3] The trickiest is the family of X_{ij} where $i < k < j$. As an intermediate step, you might show that the expected sum is $O\big(k \log(n/k)\big)$. Is this increasing or decreasing in k?

[4] There is a slick proof.

2. Adjust the randomized spanning tree algorithm to account for weights, and prove its correctness.

Exercise 4. Let $G = (V, E)$ be an undirected graph. For $k \in \mathbb{N}$ a ***k*-cut** is a set of edges whose removal disconnects the graph into at least k connected components. Note that for $k \geq 3$, the minimum k-cut problem cannot easily be reduced to (s, t)-flow. In fact, the problem is NP-Hard when k is part of the input.

1. Design and analyze a randomized algorithm that runs in $n^{O(k)}$ randomized time and returns the minimum k-cut with high probability.

2. How does your algorithm relate to the preceding statement that k-cut is NP-Hard?

Exercise 5. Consider the minimum cut problem in undirected graphs. For $\alpha \geq 1$, we say that a cut C is a ***α*-approximate minimum cut** it it's capacity is at most α times the capacity of the minimum cut.

1. Let C be an α-approximate minimum cut. Prove that the `random-contractions` algorithm returns C with probability $\geq 1/\binom{n}{\lceil 2\alpha \rceil}$.

2. Show the number of α-approximate minimum cuts is at most

$$\binom{n}{\lceil 2\alpha \rceil}.$$

References

[1] David R. Karger. "Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm". In: *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*. Ed. by Vijaya Ramachandran. ACM/SIAM, 1993, pp. 21–30.

[2] Hiroshi Nagamochi and Toshihide Ibaraki. "Computing Edge-Connectivity in Multigraphs and Capacitated Graphs". In: *SIAM J. Discret. Math.* 5.1 (1992), pp. 54–66. DOI: 10.1137/0405004. URL: https://doi.org/10.1137/0405004.

[3] Kent Quanrud. *Randomized Algorithms, Fall 2020*. 2020. URL: http://fundamentalalgorithms.com/randomized.