Randomized Reduction

Brian C. Dean, Raghuveer Mohan, Chad G. Waters
School of Computing, Clemson University
Clemson, SC, USA
{bcdean, rmohan, cgwater}@clemson.edu

ABSTRACT

Despite their power and simplicity, randomized algorithms are often under-emphasized in the classroom (and as a consequence, ultimately in practice) since they can be more challenging to analyze than their deterministic counterparts. In this paper, we describe a simplified framework that streamlines the analysis of dozens of common randomized algorithms and data structures. The key component of this framework, which we call the *randomized reduction lemma*, builds on intuition that is already commonly held by most students based on their experience with deterministic algorithms, and reduces the necessary prerequisites one must know from probability theory to a minimal subset. For example, one can prove that randomized quicksort runs in $O(n \log n)$ time with high probability in two paragraphs, without knowledge of random variables or Chernoff bounds. This paper is intended to be self-contained and written in a sufficiently student-friendly fashion so that it may serve as a classroom handout.

1. INTRODUCTION

Randomized algorithms and data structures offer many advantages over their deterministic counterparts, including speed, simplicity of implementation, resistance to malicious inputs, and overall elegance. Unfortunately, few would include "ease of analysis" in this list, since the analysis of even simple randomized algorithms can require somewhat intricate formulation, messy calculation, and prior familiarity with many tools from probability theory. The goal of this paper is to help alleviate this problem by introducing a lightweight mathematical framework that simplifies the understanding and analysis of dozens of common randomized algorithms and data structures. It requires minimal prerequisite knowledge of probability theory, and it builds on intuition that is already present in many students from their work with deterministic algorithms.

Let us start with the simple example of binary searching a sorted array $A[1 \ldots n]$ for a value v. We compare v with the value of the middle array element, $A[k]$ (with $k = n/2$), and then recursively search the left side $A[1 \ldots k-1]$ or the right side $A[k+1 \ldots n]$ as appropriate. It is easy to see that this runs in $O(\log n)$ worst-case time, owing to the following simple insight:

LEMMA 1. **(The Reduction Lemma)** *If each iteration of an algorithm reduces the effective size of a problem to at most some constant fraction $q < 1$ of its current size, then the algorithm requires $O(\log n)$ iterations.*

With binary search, each iteration reduces the effective problem size by a factor of $q = 1/2$ (e.g., from n to $n/2$). In our new framework, analysis of many randomized algorithms requires little more than the insight above. We only need to show that each iteration has reasonable likelihood of reducing the effective problem size:

LEMMA 2. **(The Randomized Reduction Lemma)** *If each iteration independently reduces the effective size of a problem to at most some constant fraction $q < 1$ of its current size with at least some constant probability $p > 0$, then we take $O(\log n)$ iterations with high probability.*

The phrase "with high probability" (whp) has come to have a fairly standard meaning in the analysis of randomized algorithms. It means that there is only a $1/n^c$ chance that our algorithm fails to achieve its target of $O(\log n)$ iterations, where c is a constant that can be set arbitrarily high by adjusting the hidden constant in the $O(\log n)$ running time bound. For example, we can take the failure probability to be something as miniscule as $1/n^{100}$ by assuming a sufficiently large constant in the $O(\log n)$ bound. As problem size n increases, a high probability guarantee becomes stronger and stronger, making it nearly certain that our algorithm will run in $O(\log n)$ iterations for large n.

By using the randomized reduction lemma, we can avoid much of the complexity inherent in analyzing randomized algorithms, such as random variables and Chernoff bounds (we show in the Appendix how to prove the randomized reduction lemma using these concepts). In some sense, it provides a higher-level interface to the world of probability theory that is specifically designed to simplify the analysis of randomized algorithms. All we need to do to apply the randomized reduction lemma is to provide suitable values for the constants p and q, as in the following example.

1.1 Randomized Binary Search

Suppose we modify binary search so that in each iteration, we compare against an element $A[k]$ with k *randomly*

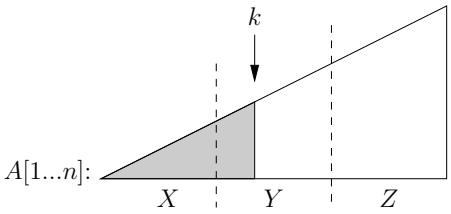

Figure 1: Analysis of randomized binary search. If we are looking for a value $v < A[k]$, then the next subproblem (shaded) will exclude all of region Z.

chosen in the range $1 \ldots n$. This may be a more accurate model for, say, a human searching a dictionary, since it is hard to manually open a book to its exact middle page. By mentally picturing the sorted array $A[1 \ldots n]$ divided into three equal-sized regions X, Y, and Z (Figure 1), we see that choice of k from the middle region Y (which happens with probability 1/3) ensures that the next iteration will involve a subproblem that either excludes all of X or all of Z, and hence will be at most 2/3 the size of the original problem. Each iteration therefore reduces the effective size of our problem to at most a constant fraction $q = 2/3$ of its original size with at least probability $p = 1/3$. According to the randomized reduction lemma, randomized binary search therefore runs in $O(\log n)$ time whp[1].

1.2 The Union Bound

A simple way we often analyze algorithms is by looking at how much time they spend on each individual input element:

LEMMA 3. *If an algorithm spends $O(f(n))$ time per input element, then it spends $O(nf(n))$ total time.*

For example, an algorithm that invests $O(\log n)$ time processing each of its n input elements takes $O(n \log n)$ total time. The *union bound* (the only other probabilistic tool we need in this paper) allows us to perform the same type of analysis for randomized algorithms:

LEMMA 4. *If an algorithm spends $O(f(n))$ time per input element whp, then it spends $O(nf(n))$ total time whp.*

For example, we will see in a moment that randomized quicksort spends $O(\log n)$ time whp on each single input element (due to the randomized reduction lemma), so it runs in $O(n \log n)$ total time whp.

To give more mathematical detail, the union bound says that if we have a set of bad events $E_1 \ldots E_n$, the probability at least one of them happens is at most the sum of their probabilities $\sum \mathbf{Pr}[E_i]$. If we let E_i be the event that we spend too much time (e.g., more than $O(\log n)$ time) on input element i, we have $\mathbf{Pr}[E_i] \leq 1/n^c$, so the probability we spend too much time on *any* element of the input is at most $\sum \mathbf{Pr}[E_i] \leq n \times 1/n^c = 1/n^{c-1}$. Since our choice of constant c can be arbitrary, this is still a high probability bound.

[1]Many students commonly ask why the simpler choice of $p = q = 1/2$ does not work for this example. We leave this as an exercise for the reader.

1.3 Randomized Quicksort

To sort an array $A[1 \ldots n]$, the popular randomized quicksort algorithm does the following:

1. Choose a random "pivot" value v from the array.
2. Partition the array in $O(n)$ time, rearranging its contents into two pieces $A[1 \ldots k]$ (containing values $\leq v$) and $A[k+1 \ldots n]$ (containing values $\geq v$).
3. Recursively sort the two pieces.

Analysis. Each partition spends $O(1)$ time per element, so the running time spent per element over the entire algorithm is essentially just the number of partitions in which the element takes part. If we can show that each element takes part in $O(\log n)$ partitions whp, the union bound therefore allows us to conclude that randomized quicksort runs in $O(n \log n)$ total time whp.

Why does each element take part in $O(\log n)$ partitions whp? If we look again at Figure 1, suppose we choose a pivot v that would have come from the middle third of our array if it were sorted. In this case, the result of partitioning on v will be two pieces each of size at most $2n/3$. Hence, each time an element takes part in a partition operation, there is at least a 1/3 probability that the element will afterwards find itself in a subproblem of at most 2/3 the size of its original enclosing array. The randomized reduction lemma therefore ensures that each element takes part in $O(\log n)$ partitions whp[2].

2. SIMPLE MATHEMATICAL EXAMPLES

To illustrate some common approaches for applying the randomized reduction lemma, let us show how it simplifies the analysis of several common probabilistic constructions we often encounter in the analysis of algorithms.

2.1 Resetting a Running Minimum

Suppose we maintain a running minimum while scanning through a randomly-ordered array $A[1 \ldots n]$. How many times does the minimum get reset? For simplicity, assume that there is a dummy "$-\infty$" at the end of the array, which becomes the final minimum. As shown in Figure 2(a), imagine following the chain of successive minima uncovered by our scan, only in reverse, starting from the final dummy element. It turns out that the indices we visit (circled in the figure) satisfy the conditions of the randomized reduction lemma with $p = q = 1/2$: if we are at index j, it is equally likely that the preceding minimum lives in $A[1 \ldots j/2]$ versus $A[j/2+1 \ldots j-1]$, owing to the fact that A is randomly ordered. Hence, there is a probability of 1/2 that our next step takes us into $A[1 \ldots j/2]$ and reduces our current index to at most 1/2 of its current value. We therefore reset our running minimum $O(\log n)$ times whp.

Applications. The property above leads to several useful algorithmic applications. Chan uses it as the basis for an elegant form of randomized divide and conquer [3], and it leads

[2]The similarity between this analysis and that of randomized binary search is no accident, since randomized quicksort "looks like" randomized binary search through the eyes of a specific element e. That is, e experiences the same comparisons as if we were searching for it with randomized binary search.

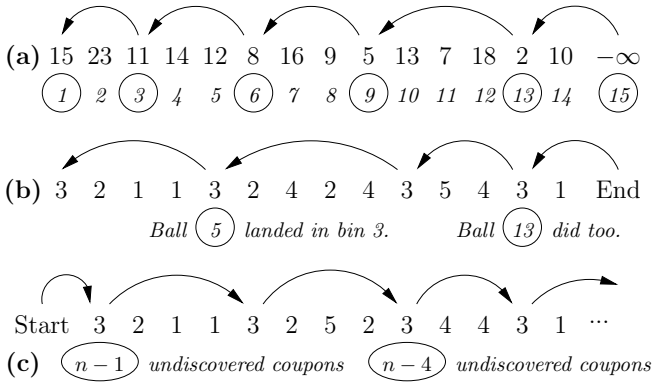

Figure 2: Randomized reduction sequences (circled) obtained from (a) resetting a running minimum while scanning through a randomly-permuted array, (b) random assignment of balls to bins, and (c) coupon collecting.

to a straightforward proof that Dijkstra's shortest path algorithm runs quickly on an n-node, m-edge graph where edge weights are drawn randomly from a common distribution; here, Dijkstra's algorithm makes only $O(n \log n)$ node label updates whp instead of the worst-case possibility of m updates [6]. As another nice example, in a set of points in the 2D plane, a point (x, y) is *non-dominated* if there is no other point (x', y') with $x' \geq x$ and $y' \geq y$. This is an important concept in multi-objective optimization, where a solution for a two-objective problem is generally not worth considering if some other solution dominates it in both objectives. A *random* set of n points has $O(\log n)$ non-dominated points whp. As shown in Figure 3, this follows by scanning the points in decreasing order of x keeping a running maximum in y. A non-dominated point occurs whenever the running maximum is reset.

2.2 Balls in Bins

Suppose we throw n balls randomly into n bins. In Figure 2(b), we have listed the sequence of n bins into which our balls have landed. If we pick a specific bin (bin 3 in the example) and visit in reverse order all the balls that land in it, the sequence of indices we visit (circled in the figure) will satisfy the randomized reduction lemma with $p = 1/4$ and $q = 0$: if we are at index j, the probability that none of the $j - 1$ preceding balls lands in our bin is $(1 - 1/n)^{j-1} \geq (1 - 1/n)^n \geq 1/4$ (for $n \geq 2$). Hence, each bin receives at most $O(\log n)$ balls whp, and by applying a union bound across all the bins, we also conclude that the fullest bin receives $O(\log n)$ balls whp.

Applications. If we need to assign computational tasks to servers, the result above tells us that reasonably good load balancing occurs if we simply assign tasks to servers at random. Another related result is that a random n-node labeled tree has maximum degree $O(\log n)$ whp. There is a one-to-one mapping between an n-node labeled tree and a so-called *Prüfer sequence* [10] of $n - 2$ integers in the range $1 \ldots n$, so the first $n-2$ elements of the length-n balls-in-bins sequence shown in Figure 2(b) codes for a unique random tree. The degree of a node x in the tree is one more than the number of occurrences of x in the Prüfer sequence, so maximum node degree corresponds to the fullest bin.

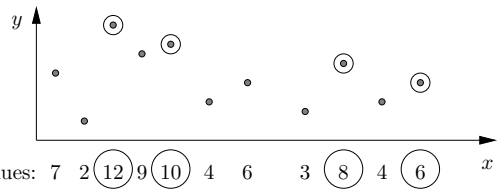

Figure 3: Counting non-dominated points (circled) in the 2D plane.

2.3 Coupon Collecting

Suppose that when we open a box of cereal, we receive (at random) one of n different coupon types. How many boxes of cereal must we plan to open before we expect to receive at least one of each coupon type? Figure 2(c) shows a random sequence of coupon types. Suppose we pick a specific coupon type (3 in this example) and imagine visiting the chain of all its occurrences, writing down at each step the number of yet-undiscovered coupon types (circled in the figure). We claim this sequence satisfies the randomized reduction lemma with $p = q = 1/2$: starting from an occurrence of coupon type c where j undiscovered coupons remain, let us consider the next $j/2$ events that are either of type I (draw a coupon of type c) or type II (draw a coupon type we have never seen before). If the next $j/2$ such events are all of type II, then the next occurrence of coupon c will correspond to a reduction in undiscovered coupon types from j to at most $j/2$, a factor of $q = 1/2$. The probability of this is

$$\left(\frac{j}{j+1}\right)\left(\frac{j-1}{j}\right)\left(\frac{j-2}{j-1}\right)\cdots\left(\frac{j/2+1}{j/2+2}\right) = \frac{j/2+1}{j+1} \geq 1/2.$$

This implies that each coupon type can occur only $O(\log n)$ times in the sequence whp, so we collect all the coupon types after opening $O(n \log n)$ boxes of cereal whp.

Applications. "Coupon collecting" is the fundamental process behind sampling with replacement, and the analysis above tells us that $O(n \log n)$ samples whp are needed to sample each of n different elements at least once. The same process shows up in many algorithmic scenarios. For example, if we perform a random walk in an n-node graph with all edges present (i.e., a complete graph), then we cover all the nodes in the graph in $O(n \log n)$ steps whp.

3. DATA STRUCTURES

Quite a few prominent randomized data structures become much easier to analyze using randomized reduction.

3.1 Randomly-Built Binary Search Trees

Since a binary search tree (BST) can be regarded as the recursion tree resulting from quicksort, there is a well-known "equivalence" between randomized quicksort and the process of building a BST by inserting its n elements in random order. This is illustrated in Figure 4(a), which depicts the tree of recursive subproblems generated by randomized quicksort (i.e., the top row is the array after partitioning on the randomly-chosen pivot 13, the next row shows partitions of the two resulting subproblems, and so on). Pivots are highlighted, and form a BST – the same tree we would have obtained by inserting elements into an initially-empty BST in the order they were chosen as pivots by randomized

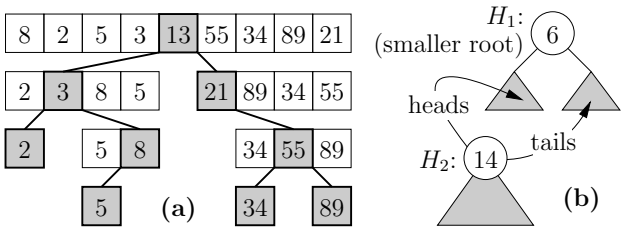

Figure 4: (a) Equivalence of binary search tree construction and quicksort, and (b) Flipping a coin to determine the subtree of heap-ordered tree H_1 into which we should recursively merge heap-ordered tree H_2.

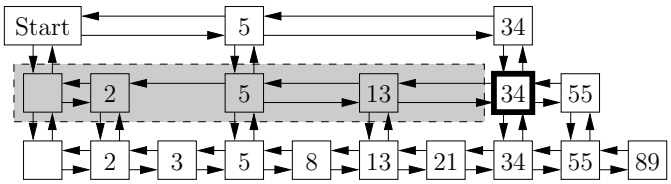

Figure 5: A skip list. The outlined node indicates our current location when traveling back to the start during an "unfind" operation.

quicksort (i.e., random order). The same pairwise comparisons between elements occur during both processes, just in different orders.

Leveraging our previous analysis of randomized quicksort, we can now say many things about randomly-built BSTs. The depth of a node in the BST corresponds to the number of partitions in which it took part, which is $O(\log n)$ whp. Taking a union bound over all nodes, we see that a randomly-built BST has depth $O(\log n)$ whp. This fact leads to several beautiful randomized balancing mechanisms for BSTs, either by making simple random adjustments during the insertion or deletion of elements [8], or by achieving a similar effect by hybridizing a BST and a binary heap to form a so-called "treap" [2]. Both approaches keep the tree always in a state which is as if it had just been built randomly from scratch, so the tree always remains balanced (i.e., $O(\log n)$ depth) whp.

3.2 Randomly-Mergeable Binary Heaps

A binary tree is heap-ordered if the value stored by a parent node is always no larger than its children. Since the minimum resides in an obvious place (the root), such trees are commonly used as priority queues, maintaining a dynamic set of elements from which we can quickly extract the minimum. If we can *merge* two heap-ordered trees into one, then from this we can build all common priority queue operations. For example, we can insert a new element by merging with a single-element tree, and we can extract the minimum (the root) by removing it and merging its two child subtrees together.

Randomization gives a fast and remarkably simple method for merging [5]. Given two heap-ordered trees H_1 and H_2, where H_1 has the smaller root, we recursively merge H_2 with one of the two subtrees of the root of H_1, determined by a coin flip, as shown in Figure 4(b). One of these two subtrees is at most half the size of H_1, so with probability $p = 1/2$, we are left with a merging subproblem where one of our two trees has reduced itself in size by a factor of $q = 1/2$. Merging (and every other fundamental priority queue operation) therefore takes $O(\log n)$ time whp.

3.3 Skip Lists

The skip list [11] was developed as a simpler alternative to the balanced BST with equivalent functionality and speed. As shown in Figure 5, it consists of increasingly-sparser copies of a linked list stacked atop each-other, with corresponding elements linked by "vertical" pointers. Each level contains roughly half the elements from the level beneath it, and this is easy to achieve using a simple randomized approach: each time a new element is inserted (initially into the lowest level), we repeatedly flip a fair coin and continue promoting it to higher levels as long as we flip "heads".

The height of a skip list is $O(\log n)$ whp due to randomized reduction, since each time we move up a level, we reduce the number of elements to at most a $q = 1/2$ fraction of the current level with probability at least $p = 1/2$. This follows directly from the fact that when we flip n fair coins, the probability that we flip at most $n/2$ heads is at least $1/2$. We employ several coin-flipping arguments of this form moving forward in this paper, all using the simple fact that after flipping n coins,

$$\mathbf{Pr}[< n/2 \text{ heads}] + \mathbf{Pr}[= n/2 \text{ heads}] + \mathbf{Pr}[> n/2 \text{ heads}] = 1.$$

By symmetry the first and last terms are equal, giving us the analysis above. Later we will deal with coins that are biased, with $\geq 1/2$ probability of landing on heads, tipping the balance between the first and last terms so $\mathbf{Pr}[> n/2 \text{ heads}] \geq \mathbf{Pr}[< n/2 \text{ heads}]$.

It takes $O(\log n)$ time whp to find an element in a skip list by starting in the upper-left corner and repeatedly moving right, only stepping down when our next rightward step would overshoot the element we seek. Insertion and deletion of elements also takes $O(\log n)$ time whp, since finding an element (or in the case of insert, finding the location in which to place a new element) is the bottleneck in both procedures. To analyze the running time for finding an element e, consider instead the equivalent running time for "unfinding" e, where we travel from e back to the start along the reversal of the path find would have taken – stepping up whenever possible, and left otherwise. As we travel, consider the number of elements directly to our left, shown shaded in Figure 5. This number satisfies the conditions of the randomized reduction lemma with $p = 1/4$ and $q = 1/2$: consider event I (our next step moves up, since we flipped heads at the current location during insertion) and II (at most half of the elements to our left flipped heads during insertion and are promoted to the next level). Both I and II are independent and happen with probability at least $1/2$ (so $p = 1/4$), and if both happen, then the number of elements to our left reduces to at most $1/2$ of the current value on our next step.

There is a straightforward one-to-one mapping between skip lists and BSTs. Using this, the mechanics of the skip list can be "ported" to obtain another simple randomized BST balancing mechanism, whose analysis follows directly from that of skip lists, and therefore also from randomized reduction [4].

Randomized reduction can also be used to show that a

skip list occupies $O(n)$ space *whp*, using another useful variation of the randomized reduction lemma:

LEMMA 5. **(Incremental Reduction)** *If an algorithm independently reduces the effective size of a problem in each iteration by 1 with at least some constant probability $p > 0$, then the algorithm takes $O(n)$ iterations whp.*

To show this, we apply the randomized reduction lemma to the quantity 2^n, for which a reduction of 1 in n corresponds to a decrease by a factor of $q = 1/2$. The original randomized reduction lemma therefore implies a bound of $O(\log 2^n) = O(n)$ iterations whp (and in fact the high probability bound here is even stronger, with a failure probability of the form $1/2^{n^c}$ instead of $1/n^c$). To apply incremental reduction to skip lists, imagine walking across the skip list from left to right, visiting each column from bottom to top along the way. Each step takes us up with probability 1/2, or to the bottom of the next column otherwise. Letting m denote the number of columns to our right, this quantity satisfies the conditions of the lemma above, so we take $O(n)$ total steps whp.

4. EXAMPLES IN NETWORKING

The domain of networking contains several applications that benefit from randomized reduction analysis.

4.1 Broadcast Along a Random Tree

We already discussed random trees briefly in Section 2.2. Here, we show how randomized reduction helps to analyze a slightly different random tree construction. Suppose we want to broadcast information from node 1 to each of nodes $1 \ldots n$ in a network. This is commonly done along a "broadcast tree" emanating outward from node 1 as its root. A simple way to choose such a tree is for each node j to select a random node in $1 \ldots j-1$ as its parent. It is easy to show that node n is only $O(\log n)$ steps away from the root whp, and since this is the worst case, the same holds true for every node, so according to a union bound over all nodes, the tree has depth $O(\log n)$ whp. The randomized reduction argument here is almost the same as with our "reset minimum" example: walk from node n back to node 1, observing that each step has a probability of $\geq 1/2$ of moving from node j to a node in the range $1 \ldots j/2$, so $p = q = 1/2$.

Another desirable feature of a broadcast tree is that each node should have low degree (or equivalently, not too many children), since otherwise it will be overloaded with outbound communication. Our random construction gives a tree with maximum degree $O(\log n)$ whp also due to randomized reduction. Consider reading off the children of some node x in decreasing order of index (e.g., the children might be nodes 17, 10, 5, and 3). If we are currently at index j, the probability that none of the nodes in the range $2j/3 + 1 \ldots j-1$ were joined to x as a parent is

$$\begin{aligned}
\left(1 - \frac{1}{2j/3}\right) \cdots \left(1 - \frac{1}{j-2}\right) &\geq \left(1 - \frac{1}{2j/3}\right)^{j/3-1} \\
&\geq \left[\left(1 - \frac{1}{2j/3}\right)^{2j/3}\right]^{1/2} \\
&\geq [1/4]^{1/2} \quad (\text{for } j \geq 3) \\
&= 1/2,
\end{aligned}$$

so we reduce our index from j by a factor of $q = 2/3$ to a number in $1 \ldots 2j/3$ with probability at least $p = 1/2$.

4.2 Contention Resolution

Suppose n devices are all trying to transmit along a shared communication channel, where in each timestep a device can choose whether or not to attempt transmission, but if multiple devices transmit, they will interfere (and they can detect when this happens). Suppose the devices do not know n, so they do not know how many other devices they are competing against. Here, a reasonable protocol for contention resolution is for each device to flip a coin and attempt transmission if heads. If transmission fails due to a collision, however, all transmitting devices enter a dormant state until the first timestep in which no transmission occurs, then they wake up and resume the same protocol. In the first timestep, roughly half of our devices will attempt transmission and become dormant, followed by roughly a quarter in the next timestep, and so on.

In terms of randomized reduction, a simple coin flip argument tells us that each step reduces the number of active devices by at least a factor of $q = 1/2$ with probability at least $p = 1/2$, so we will have $O(\log n)$ timesteps whp until the end of a "phase", which can be either a successful transmission by one device or a timestep with no transmissions (e.g., if all remaining devices had attempted transmission and gone to sleep). Fortunately, a straightforward argument can show that each phase ends with a successful transmission with probability at least 1/2, giving a reasonable bound on the frequency of successful transmissions.

4.3 Gossip Protocols

Suppose a machine wants to transmit a message to n other machines in a distributed network. A simple protocol for doing this is the following: in each timestep, every machine that knows the message transmits it to another machine randomly chosen from $1 \ldots n$ (if unlucky, this machine already knows the message, in which case the transmission is wasted). This is a "pushing" variant of a so-called *gossip* protocol; we can similarly conceive of "pulling" variants where machines without the message ask for it from random machines in each timestep.

Randomized reduction gives a nice way to analyze the number of timesteps necessary for a gossip protocol to spread its message to all n machines. In the push variant, for example, let us divide time into two phases: the first phase runs until $\geq n/4$ machines have heard the message, and then the second phase starts. In each timestep of the first phase, suppose we have $k < n/4$ machines with knowledge of the message. Each one pushes to a random machine, and since at least $n/2$ machines don't have the message (even taking into account new machines hearing the message in this timestep), each push succeeds with probability $\geq 1/2$. We can therefore regard this as flipping k coins that are biased to land on heads with probability $\geq 1/2$, so we have $\geq k/2$ heads (successful pushes) with probability $\geq 1/2$. Therefore, we have a probability of at least $p \geq 1/2$ of expanding the number of machines k with the message by a factor of 3/2 (or equivalently, of reducing the quantity n/k by a factor of $q = 2/3$). Hence, the first phase completes in $O(\log n)$ timesteps whp.

In the second phase, a direct probability argument suffices. Consider a machine m that doesn't know the message. In

each timestep, since $k \geq n/4$, the probability none of our k machines chooses m is $(1-1/n)^k \leq (1-1/n)^{n/4} \leq e^{-1/4}$ (owing to the useful bound $(1-1/n)^n \leq e^{-1}$). If we run for $c \ln n$ timesteps, the probability m doesn't hear the message during any timestep is therefore at most $e^{-c(\ln n)/4} = 1/n^{c/4}$. This gives a high probability bound of success for machine m in $O(\log n)$ timesteps, which after a union bound then implies a high probability of success for all machines in $O(\log n)$ timesteps.

5. CONCLUSIONS AND EXTENSIONS

We have shown how the principle of randomized reduction leads to short, simple analyses for a broad number of computational results. For space considerations, there were many more examples we could not mention, including analysis of standard algorithms such as interpolation search [9] and stable in-place randomized quicksort [12], parallel and distributed algorithms (e.g., parallel maximal independent set [1, 7]), algorithms for problems in computational geometry (e.g., convex hull), and methods from other domains as well. We hope this technique helps promote greater adoption of randomized algorithms in computing classes, and ultimately in real-world systems.

6. REFERENCES

[1] Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7(4):567 – 583, 1986.

[2] Cecilia R. Aragon and Raimund Seidel. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[3] Timothy M. Chan. Geometric applications of a randomized optimization technique. In *Proceedings of the Symposium on Computational Geometry*, pages 269–278, 1998.

[4] Brian C. Dean and Zachary H. Jones. Exploring the duality between skip lists and binary search trees. In *Proceedings of the 45th ACM Southeast Conference (ACMSE)*, pages 395–399, 2007.

[5] Anna Gambin and Adam Malinowski. Randomized meldable priority queues. In *Conference on Current Trends in Theory and Practice of Informatics*, pages 344–349, 1998.

[6] Andrew V. Goldberg and Robert E. Tarjan. Expected performance of Dijkstra's shortest path algorithm. Technical Report TR-96-063, NEC Research Institute, 1996.

[7] Michael Luby. Simple parallel algorithms for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.

[8] Conrado Martínez and Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998.

[9] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search — A log log N search. *Communications of the ACM*, 21(7):550–553, 1978.

[10] Heinz Prüfer. Neuer beweis eines satzes über permutationen. *Archiv für Mathematik und Physik*, 27:142–144, 1918.

[11] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[12] Ronald L. Rivest. A fast stable minimum storage sorting algorithm. Technical Report Rep. 43, Institute de Recherche d'Informatique, Rocquencont, France, 1973.

APPENDIX

A. PROOF OF THE RR LEMMA

For completeness, we now show how to prove the randomized reduction lemma using other tools from probability theory. We also prove that the randomized reduction lemma can give an $O(\log n)$ bound in expectation as well as with high probability, and that it applies also if we can show a reduction in expected problem size during each iteration.

PROOF. Let T be the running time of an algorithm satisfying the conditions of the randomized reduction lemma. If we run just for the first $L = k \ln n$ iterations, let X be the number of these iterations on which we see adequate reduction (by a factor of q). Since reduction in each iteration happens with probability at least p, $\mathbf{E}[X] \geq Lp$. If our algorithm fails to terminate in L units of time, then it must be the case that $X \leq \log_{1/q} n$, since otherwise we would have had enough reducing iterations to go from an initial problem size n down to 1. Hence, $\mathbf{Pr}[T \geq L] \leq \mathbf{Pr}[X \leq \log_{1/q} n]$. Since X has a binomial distribution, we can bound the probability of deviation from its mean using a standard form of the Chernoff bound: for $\alpha \in [0, 1/2]$,

$$\mathbf{Pr}[X \leq \alpha \mathbf{E}[X]] \leq e^{-(1-\alpha)^2 \mathbf{E}[X]/2} \leq e^{-E[X]/8}$$

Setting $\alpha = \frac{1}{k} \log_{1/q} e$ and taking $k \geq 2 \log_{1/q} e$, we therefore have

$$\begin{aligned} \mathbf{Pr}[T \geq L] &\leq \mathbf{Pr}[X \leq \log_{1/q} n] \\ &\leq \mathbf{Pr}[X \leq \alpha \mathbf{E}[X]] \\ &\leq e^{-k(\ln n)/8} \\ &= 1/n^{k/8}, \end{aligned}$$

so we run in $k \ln n = O(\log n)$ iterations whp. □

COROLLARY 6. *If an algorithm satisfies the conditions of the randomized reduction lemma, it runs in $O(\log n)$ expected iterations.*

PROOF. Call a "phase" a set of iterations up to a reducing iteration; we expect each phase to contain $\leq 1/p$ iterations, since each iteration is reducing with probability $\geq p$. After $\log_{1/q} n$ phases the algorithm must have terminated, so by linearity of expectation, the total time on all phases is at most $\frac{1}{p} \log_{1/q} n = O(\log n)$. □

COROLLARY 7. *If each iteration of an algorithm reduces the expected size of our problem to at most a $q < 1$ fraction of its current value with probability at least $p > 0$, then the algorithm satisfies the conditions of the randomized reduction lemma.*

PROOF. Let $q' = (1+q)/2 < 1$, and let X be a random variable indicating the fraction of reduction on a given iteration. According to Markov's inequality, $\mathbf{Pr}[X > q'] \leq \mathbf{E}[X]/q' \leq q/q'$, so $p' = \mathbf{Pr}[X \leq q'] \geq 1 - q/q' = 1/q' - 1 > 0$. We therefore satisfy the randomized reduction lemma with parameters p' and q'. □