CS 577: Introduction to Algorithms Fall 2020

Randomness

Instructor: Dieter van Melkebeek

These notes briefly review a few additional ideas that are useful to an algorithms designer. In particular, we cover randomness in computation and some of its applications in algorithms design.

Randomness in computation is the idea of allowing computations to use coin flips to make decisions. At first glance, this may seem silly or useless; how can flipping a coin that has nothing to do with the computational problem help solve the problem? The answer to this is that, in some situations, there are many good decisions, but singling out any particular good decision may be difficult. However, by making a decision at random, there is a high chance that a good decision is made.

This approach still has a couple drawbacks: First, if there is even a single bad decision, then there is no way to avoid the possibility that we select a bad decision. Even if we select many candidates and consider them all, there is still no way to avoid the possibility that every choice happened to be a bad choice. Thus there will always be an element of error in this approach. Second, there is the issue of procuring randomness.

The first issue we can address by insisting that the probability of errors be extremely small. For most situations, it suffices to repeat the random sampling procedure many times, and then combine the results in some natural way. For instance, if there is an easy way to tell *whether* a decision is good, then choosing many random decisions and picking the first one that is good will succeed with high probability.

The second issue is mostly beyond the scope of these notes, but it is worth pointing out that randomness is not, in general, a cheap resource. In most applications, the randomness required is often small enough not to matter, or can be replaced by a pseudorandomness generator; however, in some settings (*e.g.*, generation of secrets in cryptography), it's crucial that a lot of real randomness is used. For settings like these, it's essential that the use of randomness is kept small.

However, despite these drawbacks, randomness can be a great boon to computation. These notes present a few high-level examples for demonstration's sake. We start with a small introduction to the required probability theory before moving into the examples.

1 Basic Probability Theory

A *probability space* is given by its *universe* Ω of underlying events and by a distribution $\Pr : \Omega \to [0,1]$. Pr satisfies the following equation:

$$\sum_{\omega \in \Omega} \Pr[\omega] = 1$$

$\Pr[\omega]$ indicates the *probability* that the event ω happens.

An example is that of a single fair coin flip: in this case, Ω can be regarded as the set $\{\texttt{Heads}, \texttt{Tails}\}$, and the distribution is given by $\Pr[\texttt{Heads}] = \Pr[\texttt{Tails}] = 1/2$.

A *random variable* is a function $X : \Omega \to \mathbb{R}$. We can think of random variables as quantifying some information about an underlying event in Ω. For instance, to continue the coin flip analogy, we might consider the random variable H defined by $H(\texttt{Heads}) = 1$ and $H(\texttt{Tails}) = 0$. One could of this is as saying "H is one if the coin lands on `Heads`, and is zero if the coin lands on `Tails`". One might also think of H as counting the number of heads that occur, or indicating that a heads is the result of the coin flip.

Random variables have *expected* (or *average*) *values*. The expectation of a random variable X is denoted by $\mathrm{E}[X]$ and is defined to be the value

$$\mathrm{E}[X] \doteq \sum_{\omega\in\Omega} \Pr[\omega] \cdot X(\omega)$$

Since the sum of all values of $\Pr[\omega]$ is one, we can view this as taking a weighted average of the possible values of X. In our example with the coin flip, we can compute

$$\mathrm{E}[H] = \Pr[\texttt{Heads}] \cdot 1 + \Pr[\texttt{Tails}] \cdot 0 = 1/2$$

An important inequality involving expectations is Markov's inequality:

Lemma 1. *For any non-negative random variable X and any real number $a \geq 0$,*

$$\mathrm{E}[X] \geq a \cdot \Pr[X \geq a]$$

If $\mathrm{E}[X] > 0$ and $a > 0$, this can be rearranged to

$$\Pr[X \geq a \cdot \mathrm{E}[X]] \leq \frac{1}{a}$$

Proof. The basic idea is to lower bound the value of $\mathrm{E}[X]$ by first throwing away all the contributions from events when $X < a$, and then using that $X \geq a$ on the remaining events. Algebraically, we realize this as

$$\begin{aligned}
\mathrm{E}[X] &\doteq \sum_{\omega\in\Omega} \Pr[\omega] \cdot X(\omega) \\
&= \sum_{\omega\in\Omega : X(\omega) < a} \Pr[\omega] \cdot X(\omega) + \sum_{\omega\in\Omega : X(\omega) \geq a} \Pr[\omega] \cdot X(\omega) \\
&\geq 0 + \sum_{\omega\in\Omega : X(\omega) \geq a} \Pr[\omega] \cdot a \\
&= a \cdot \Pr[X \geq a]
\end{aligned}$$

To get the re-arranged inequality, assume $\mathrm{E}[X] > 0$ and let $a' > 0$ be given. let $a = a' \cdot \mathrm{E}[X]$. Applying the inequality pre-rearrangement, we get

$$\mathrm{E}[X] \geq a' \cdot \mathrm{E}[X] \cdot \Pr[X \geq a' \cdot \mathrm{E}[X]]$$

which can be re-arranged to yield the second inequality in the statement of the lemma. □

2 Randomized Selection

Recall the specification of the selection problem:

Input: an array A of n numbers, and an integer k with $1 \leq k \leq n$.

Output: the kth element of A when sorted in non-decreasing order.

In the last lecture on divide-and-conquer we developed a deterministic linear-time algorithm for this problem. It uses an approximate median of A as a pivot p to break up the array A into the subarray L_p of all entries with value less than p, the entries equal to p, and the subarray R_p of all entries larger than p. Given p, we can construct L_p and R_p in linear time, and based on their sizes we know whether the kth smallest element of A lies in L_p, equals p, or lies in R_p. In the middle case, we're done; in the other two cases we recurse on L_p (with the same value of k) or on R_p (with the value of k reduced by $n - |R_p|$), respectively

The procedure for finding an approximate median was somewhat complicated and involved another recursive call. Instead of doing that, we now simply pick the pivot p uniformly at random among the elements of A. Intuitively, such a random pivot has a good probability of being an approximate median in the sense of reducing the size of the remaining array by a constant factor less than one. As such a reduction in size can only happen $O(\log n)$ times, chances are the recursion bottoms out within $O(\log n)$ levels. The amount of local work at each level would roughly behave like a geometric sequence with ratio less than one, resulting in a linear overall expected running time. We now formalize this intuition.

For the purposes of the analysis, we refer to the elements of the array A by their index in the sorted order, and we think of picking the pivot p as follows: take a real r in the interval $(0, n)$ uniformly at random, and round up to the next integer. Note that the distribution of p is indeed uniform over $\{1, \ldots, n\}$, over the entries of A.

For any integer $i \geq 0$, consider the following random variable:

$$X_i = \begin{cases} \text{size of the subarray at the } i\text{th level of recursion} & \text{if the } i\text{th level exists} \\ 0 & \text{otherwise.} \end{cases}$$

We consider the original call to the procedure to be the 0th level of recursion, so $X_0 = n$. Since the amount of work we spend at the ith level of recusion is at most $c \cdot X_i$ for some constant c, the total amount of work is at most $c \cdot \sum_i X_i$.

Claim 1. *(Key Claim) For every integer $i \geq 0$,*

$$\mathrm{E}[X_{i+1}] \leq \frac{3}{4} \cdot \mathrm{E}[X_i]. \tag{1}$$

Proof. We first show the following for every fixed integer ℓ:

$$\mathrm{E}[X_{i+1} \,|\, X_i = \ell] \leq \frac{3}{4}\ell. \tag{2}$$

Since $X_{i+1} = 0$ whenever $X_i = 0$, we only need to consider cases with $\ell > 0$. In order to pick the pivot p at level i, we take a real r in $(0, \ell)$ uniformly at random, and set $p \doteq \lceil r \rceil$. Note that $L_p = \{1, \ldots, p-1\}$ and $R_p = \{p+1, \ldots, \ell\}$. It follows that $|L_p| \leq r$ and $|R_p| \leq \ell - r$. Since

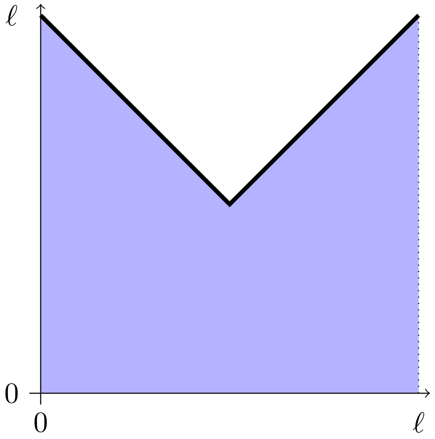


Figure 1: Plot of $\max(r, \ell - r)$ for $r \in (0, \ell)$.

The value of $\mathrm{E}[\max(r, \ell - r)]$ is given by the average height of the darkest line above, which is equal to $\frac{1}{\ell}$ times the shaded region below the line. This works out to $\frac{3}{4}\ell$.

$X_{i+1} \leq \max(|L_p|, |R_p|)$, the LHS of (2) is at most the expected value of $\max(r, \ell - r)$ when r is picked uniformly at random from $(0, \ell)$. As can be gleaned from the plot below, the latter expected value equals $\frac{3}{4}\ell$.

This follows because on the first half of the interval, i.e., for $r \in (0, \frac{\ell}{2})$, $\max(r, \ell - r) \equiv \ell - r$, a linear function that evolves from ℓ to $\frac{\ell}{2}$ on the interval $(0, \frac{\ell}{2})$, and averages $\frac{1}{2} \cdot (\ell + \frac{\ell}{2}) = \frac{3}{4}\ell$ on that interval. Similarly, on the second part of the interval, i.e., for $r \in (\frac{\ell}{2}, \ell)$, $\max(r, \ell - r) \equiv r$, and the average of r for $r \in (\frac{\ell}{2}, \ell)$ also equals $\frac{3}{4}\ell$. Thus, the overall average equals $\frac{3}{4}\ell$ as well. This establishes (2).

We obtain equation (1) from equation (2) as follows:

$$\mathrm{E}[X_{i+1}] = \sum_{\ell} \Pr[X_i = \ell] \cdot \mathrm{E}[X_{i+1} \mid X_i = \ell] \leq \sum_{\ell} \Pr[X_i = \ell] \cdot \frac{3}{4} \cdot \ell = \frac{3}{4} \cdot \mathrm{E}[X_i].$$

□

The Key Claim has the following implications.

Lemma 2. *For every integer $i \geq 0$,*

$$\mathrm{E}[X_i] \leq \left(\frac{3}{4}\right)^i \cdot n. \tag{3}$$

Proof. Repeated application of (1) shows that $\mathrm{E}[X_i] \leq \left(\frac{3}{4}\right)^i \cdot \mathrm{E}[X_0]$. Since $X_0 = n$, Lemma 2 follows. □

Theorem 1. *The expected running time of randomized selection is $O(n)$.*

Proof. The expected running time is upper bounded by $c \cdot \sum_i \mathrm{E}[X_i]$. By Lemma 2, the latter sum is no more than $\sum_i \left(\frac{3}{4}\right)^i \cdot n = 4n$. □

Lemma 3. *The probability that randomized selection has more than* $s \doteq \lceil \log_{\frac{4}{3}}(n) \rceil + \Delta$ *levels of recursion is at most* $\left(\frac{3}{4}\right)^{\Delta}$.

Proof. Randomized selection has more than s levels of recusion iff $X_s > 0$. As X_s is a non-negative random variable that only takes integer values, Markov's inequality and Lemma 2 show that

$$\Pr[X_s > 0] = \Pr[X_s \geq 1] \leq \mathrm{E}[X_s] \leq \left(\frac{3}{4}\right)^{s} \cdot n \leq \left(\frac{3}{4}\right)^{\Delta}.$$

□

Theorem 2. *The expected number of levels of recursion of randomized selection is* $O(\log n)$.

Proof. The expected number of levels of recursion can be written as

$$\begin{aligned}
&\sum_{s=1}^{\infty} \Pr[\text{ the number of levels is at least } s \text{ }] \\
&\quad = \sum_{s=0}^{\infty} \Pr[\text{ the number of levels is more than } s \text{ }] \\
&\quad = \sum_{s=0}^{\infty} \Pr[X_s > 0] \\
&\quad \leq \lceil \log_{\frac{4}{3}}(n) \rceil + \sum_{\Delta=0}^{\infty} \left(\frac{3}{4}\right)^{\Delta} = \lceil \log_{\frac{4}{3}}(n) \rceil + 4 = O(\log n),
\end{aligned}$$

where the inequality follows from breaking up the sum into:

- the terms with $s < \lceil \log_{\frac{4}{3}}(n) \rceil$, each of which is a probability and can therefore be upper bounded by 1, and
- the terms with s of the form $s = \lceil \log_{\frac{4}{3}}(n) \rceil + \Delta$ for non-negative integer Δ, each of which can be upper bounded by $\left(\frac{3}{4}\right)^{\Delta}$ by Lemma 3.

□

3 Randomized Quicksort

Recall the specification of the sorting problem:
Input: an array A of n numbers.
Output: Sort(A), i.e., the array A sorted in non-decreasing order.

We consider randomized quicksort:

1. Pick a pivot p uniformly at random among the elements of the array A.
2. Break up the array A into the subarray L_p of all entries with value less than p, the entries equal to p, and the subarray R_p of all entries larger than p.

3. Recursively sort L_p and R_p.

4. Return the concatenation of the sorted version of L_p, the entries equal to p, and the sorted version of R_p.

This algorithm always outputs the correctly sorted array A, but the running time is a random variable depending on the choices of the pivots. As in randomized selection, the local amount of work associated with a given node in the recursion tree is linear in the size of the corresponding subarray. Since the subarrays at a given level of recursion are disjoint, this implies that the amount of work per level of recursion can be upper bounded by $c \cdot n$ for some constant c. Thus, all that remains is to analyze the number of levels of recursion, which is a random variable depending on the choice of pivots.

In order to do so we observe that the number of levels of recursion for randomized quicksort on input A equals the maximum over all $k \in \{1, \ldots, n\}$ of the number of levels of recursion of randomized selection on input A and k. This observation allows us to use our analysis of randomized selection and derive the following.

Lemma 3: The probability that randomized quicksort has more than $s \doteq 2\lceil \log_{\frac{4}{3}}(n) \rceil + \Delta'$ levels of recursion is at most $\left(\frac{3}{4}\right)^{\Delta'}$.

Proof: By the above observation, randomized quicksort has more than s levels of recursion on input A iff for some $k \in \{1, \ldots, n\}$ randomized selection on input A and k has more than s levels of recursion. By Lemma 2, for any fixed $k \in \{1, \ldots, n\}$, the probability that randomized selection on input A and k has more than s levels of recursion is no more than $\left(\frac{3}{4}\right)^{\Delta}$, where $\Delta = \lceil \log_{\frac{4}{3}}(n) \rceil + \Delta'$. By a union bound over all n possible values of k, the probability that randomized quicksort on input A has more than s levels of recursion is no more than $n \cdot \left(\frac{3}{4}\right)^{\Delta} \leq \left(\frac{3}{4}\right)^{\Delta'}$. □

In the same way that we derived Theorem 2 from Lemma 2, we obtain the following result from Lemma 3.

Theorem 3: The expected number of levels of recursion of randomized quicksort is $O(\log n)$.

As the amount of work per level of recursion can be bounded by $c \cdot n$, we conclude:

Theorem 4: The expected running time of randomized quicksort is $O(n \log n)$.

Finally, we point out that the randomized process induced by running randomized quiksort on the fixed array $A = (1, 2, \ldots, n)$ is the same as running a deterministic version of quicksort on a random permutation of A. In a deterministic version of quicksort the pivot is chosen in a deterministic way; several variants exist: the first element of the array, the last one, the middle one, etc. Theorem 4 therefore also shows that the average complexity of deterministic quicksort on a random input array is $O(n \log n)$.

4 Hashing

The following sort of problem arises in many places in computer science. There is some enormous universe Ω, together with some subset $S \subseteq \Omega$ which is considerably smaller than Ω. We are interested in representing S so that we can perform the following operations:

- Decide whether a given element of Ω belongs to S

- ○ Add an element to S
- ○ Remove an element from S

If we optimize purely for time efficiency, then we could use a boolean array indexed by Ω in which the i-th entry encodes whether the i-th element of Ω is in S. This lets us lookup whether or not a given element of Ω is in S very quickly, and also lets us manage updates quickly. However, this is wildly impractical in reality, because it requires us to store one bit per element of Ω. In our application, Ω is huge—too big even to have one bit per element.

If we optimize in the other direction and go purely for space efficiency, we can just maintain a list of the elements of S. The space required is only as big as S itself, which is the best we can hope for. However, our updates now can take more than constant time to implement. We can employ a balanced binary search tree to keep this cost low—only $\Theta(\log(|S|))$ time per operation.

It turns out that we can do even better, and in fact achieve the "best of both worlds" through the use of randomness. The technique is referred to as *hashing*. The idea of hashing is as follows: suppose we have a way of labeling the elements of Ω with numbers $1, 2, \ldots, m$, where we will think of m as being a little larger than $|S|$. Many elements of Ω will have the same label; this is fine. Call the labeling f, so that the label of x is $f(x)$.

The ideal choice of f is so that every element of S gets a distinct label—in this case, we say f has no *collisions*. We then make an array A of size m, and store at position i either the unique element of S with label i, or else some indication that no element of S has label i. Storing things this way works great: to decide if a given element x of Ω is in S, we compute $f(x)$ and then examine $A[f(x)]$. If $A[f(x)] = x$, then x must be in S; otherwise, it is not. We can similarly add and remove elements from S in time independent of the size of S.[1] Overall, the space usage is m bits, which is roughly the size of S.

Thus the difficulty in implementing this data structure is in choosing a good hash function, f. The intuition is that, if we pick f *at random*, then there is a good chance that f will have no collisions. The problem with this is that it's prohibitively expensive to choose f completely at random—we would need $\Theta(|\Omega|)$ space just to store it.

However, we don't need a completely random f; it will suffice to just keep the probability of collisions low. Specifically, we want that, for any two elements a and b of Ω, the probability that a and b collide in the randomly chosen hash function is the same as if we had chosen a hash function completely at random. This latter probability is $1/m$.

A family of hash functions with this property is called a *universal family of hash functions*. This is a family $\mathcal{H} = \{h : \Omega \to [m]\}$ of labelings so that drawing a function from $\mathcal{H}$ at random is functionally the same as choosing a labeling uniformly at random from all possible labelings. Formally, we require that, for any two distinct elements a and b of Ω, when we select a labeling h from $\mathcal{H}$ at random, the probability that a and b collide under h is $1/m$. Symbolically, we require that for every two distinct elements a and b of Ω,

$$\Pr_{h \sim \mathcal{H}}[h(a) = h(b)] = \frac{1}{m}$$

An example of a family of hash functions is

$$\mathcal{H} \doteq \{h_{a,b} \mid a, b \in \{1, 2, \cdots, n\}\}$$

[1] We are thinking of $m \approx |S|$, but $|S|$ may change as addition and removal operations are performed over time. In practice, we will just pick new values of m and f whenever $|S|$ changes too much, and recompute the whole data structure. This is known as rehashing. We won't go into much detail about it.

where $h_{a,b}(x) = (ax + b) \ \% \ n$, where "$\% \ n$" means reduction modulo n (as in most programming languages). If n is prime, the above family is universal—this essentially boils down to the fact that two distinct points determine a line, but we leave the details as an exercise. The importance of this example comes from how simple it is to use: it requires only $2 \log_2(n) + O(1)$ random bits to select an element of $\mathcal{H}$ at random, and it's quite easy to come $h_{a,b}(x)$ for any given x.

Before moving on to a more involved example, we present one last fact about universal families of hash functions:

Proposition 1. *For any set $S \subseteq \Omega$ and any element u in S,*

$$\mathrm{E}[\# \textit{ collisions in } S \textit{ with } u] = \frac{|S| - 1}{m}$$

Proof. We write

$$\begin{aligned}
\mathrm{E}[\# \text{ collisions in } S \text{ with } u] &= \sum_{u' \in S, u' \neq u} \Pr_{h \sim \mathcal{H}}[h(u') = h(u)] \\
&= (\# \text{ elements in } S \text{ not equal to } u) \cdot \frac{1}{m} \\
&= \frac{|S| - 1}{m}
\end{aligned}$$

□

A simple corollary of this is that, if $m \geq 100 \cdot |S|$, then the expected number of elements of S that collide with a particular element u of S is $1/100$. Applying Markov, we can further conclude that, for any given element of S, the probability that it collides with *any* other element of S is at most 1%.

4.1 Closest Pair of Points in the Plane via Hashing

As an application of hashing, we will re-solve the problem of finding a closest pair of points in the plane. Recall the formal specification below:

Input: A finite set of points $X = \{p_1, p_2, \cdots, p_n\}$ in the plane.

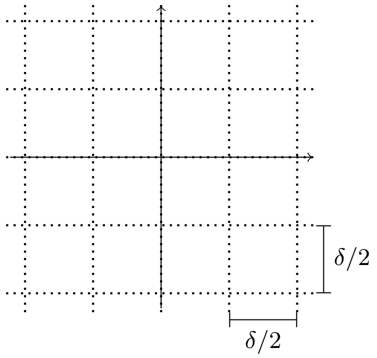
Output: The smallest distance between a pair of points of X.

We saw a deterministic divide-and-conquer algorithm for solving this problem that computed the distances between $O(n \log(n))$ pairs of points. Using randomness (specifically, hashing), we can get an intuitively simpler algorithm which makes only $O(n)$ distance computations in expectation.

The basic idea is to work in steps. Initially, we just think about the points p_1 and p_2. In the following steps, we will consider one more point—p_3, then p_4, etc.—until we are considering all n points. In each step, we will keep track of the distance between the closest pair of points, δ. The goal then is just to figure out how to compute an updated value of δ with each step.

To do this, suppose we are in the step of adding p_{i+1} to the set of points under consideration. We will think of breaking the plane up into a grid of width $\delta/2$, as in Figure 2. Doing this has two advantages. First, when we consider p_{i+1}, it belongs to a unique cell C. There are $O(1)$ cells that are within distance δ of C, *i.e.*, there are $O(1)$ cells that could conceivably contains points with

Figure 2: Breaking the plane up into a grid



The plane is broken up into a square grid of width $\delta/2$. This guarantees that, when we begin the step of adding p_{i+1} to the points $p_1, \ldots, p_i$, every cell in the grid contains at most one point among $p_1, \ldots, p_i$.

distance less than δ from p_{i+1}. Second, each cell can contain at most one point from among the points $p_1, \ldots, p_i$.

Putting these together, we can get a high-level algorithm: when adding p_{i+1} to the points in consideration, find the cell C that p_{i+1} belongs to, look at the cells within distance δ of C, and compute the distance between p_{i+1} and whichever one of the points $p_1, \ldots, p_i$ is in each cell. If we had a way to specify a cell and get back in constant time the point among $p_1, \ldots, p_i$ lying in that cell, this would be a complete linear-time algorithm.

Ordinarily, this isn't possible to do, but this is where hashing comes in. We think of Ω as consisting of all cells in the plane, and S as cells containing one of $p_1, \ldots, p_i$. We'll set the value of m to be $100n$, and initialize an array P of empty lists of size m. When we choose a hash function h, it gives a way of mapping cells to indices into P; we think of $P[x]$ as storing the points in cells with hashed value x. Using a universal family of hash functions, the probability of collisions is rare, so it's unlikely that a given entry in P will contain more than one point. Hence when we go to look up the point in a cell C, we compute $h(C)$, and then look at all the points in $P[h(C)]$.

There is some additional complication here: when we update δ, our entire hashing structure breaks. To fix this, we will need to rebuild everything, or *rehash* for short. We can re-use P so that the total cost of doing this is only the number of points under consideration in the current step.

This (very roughly) specifies the whole algorithm. In summary, we start by considering the points p_1 and p_2, initializing the value of δ, the hash function h, and the lookup table P accordingly. Then for $i = 2, 3, \ldots, n-1$, we add p_{i+1} to the points under consideration. For each step, we find the cell C containing p_{i+1}, and use our hashing structures to compute the distance between p_{i+1} and any points in the cells within δ from C. If no point is closer than δ, we add p_{i+1} to the appropriate place in P. If some point is closer than δ, we update δ and rehash. At the end, we output δ.

The running time here can be tallied up as follows:

Lookups: Since a particular cell has at most $O(1)$ cells within distance δ, each step in the above algorithm makes at most $O(1)$ lookups within P. (Somehow there are $O(1)$ many comparisons per lookup in expectation?)

Non-rehashing updates: Again linear.

Rehashing: Each time we rehash after considering k points, we have to pay a cost proportional to k. It's possible that we consider the points in a bad order, and have to rehash with each newly considered point, leading to a total cost of $O(n^2)$ for rehashing.

However, if we randomly order the input points, then rehashing turns out to be rare. Specifically, when considering the $i+1$-st point, the probability of rehashing is equal to the probability that the $i+1$-st point belongs to the closest pair of points, which is proportional to $1/i$. Since the cost of rehashing is proportional to i, we get that each potential rehash, in expectation, costs $O(1)$, and so the total cost of rehashing is $O(n)$.

5 Polynomial Identity Testing

Our final example of the usefulness of randomness in computation is that of polynomial identity testing. In this problem, the input is two polynomials, p and q. They are written as some multiplications and additions of variables and constants, *e.g.*, $(x-y)\cdot(x+y)$ or $x\cdot x-y\cdot y$. The goal is to decide whether p and q are the same polynomial, which is to say that they have the same coefficients when expanded out as a sum of monomials. For instance,

$$(x-y)\cdot(x-y) = x\cdot x - y\cdot x - x\cdot y + y\cdot y = x\cdot x + y\cdot y$$

and

$$(x+y)\cdot(x+y) - 2\cdot x\cdot y = x\cdot x + y\cdot y$$

give two different ways of writing the same polynomial.

One strategy is the naïve one: just expand p and q into a sum of monomials, and verify that the coefficients agree. However, this can take exponential time, because there are short ways of writing polynomials with many monomials. For instance, the polynomial

$$(x_1+x_2)(x_3+x_4)\cdots(x_{2n-1}+x_{2n})$$

has 2^n monomials when it is expanded out.

For this problem, no efficient deterministic procedure is known. However, there is a very simple randomized procedure that works well. The idea is to simply plug a random assignment to the variables. More formally, we choose some finite set S of integers, and randomly assign an element of S to each variable to get an assignment $x \leftarrow a$. If S is large enough and p and q are distinct polynomials, then with high probability $p(a) \neq q(a)$. The formal statement is given as Lemma 4.

Lemma 4. *(Schwartz-Zippel Lemma)*

Let p be any non-zero polynomial of degree d, and S a finite set of integers. Let $x \leftarrow a$ be an assignment to the variables of p where each component of a is chosen uniformly and independently at random from S. Then the probability that $p(a) = 0$ is at most $\frac{d}{|S|}$.

If p is a polynomial given as some formula involving m multiplications, then the degree of p is at most $m+1$. Hence if we choose $S = \{1, 2, \cdots, 2m+2\}$, then the random assignment strategy succeeds with probability at least a half. Probability 1/2 may not seem like much, but by independently repeating the test t times, the probability that $p(a) = 0$ every time is at most $\frac{1}{2}^t$, unless p is the zero polynomial.

This problem may seem a bit contrived—what does testing equality of polynomials have to do with any "real" problems? The answer is that it is used as a primitive in many routines in computer science. For instance, if one wants to solve maximum bipartite matching, then the algorithm we know uses network flow. Network flow doesn't parallelize well. However, there is a reduction from maximum bipartite matching to polynomial identity testing. Since plugging points into a polynomial can be parallelized in an obvious way, we then get an efficient randomized parallel algorithm for maximum bipartite matching.