NOI.PH Training: Algorithms 3

Floating Point Numbers and Randomized Algorithms

Cisco Ortega

Contents

1 Floating Point Numbers

Let's begin with a traditional first foray into floating point errors. Try running the following snippet and see what the output will be.

```
cout << fixed << setprecision(20); // set cout to always print 20 decimal places
cout << (0.5 + 0.5) << endl;
cout << (0.5 + 0.5 == 1.0) << endl;
cout << (1.0 + 2.0) << endl;
cout << (1.0 + 2.0 == 3.0) << endl;
cout << (0.1 + 0.2) << endl;
cout << (0.1 + 0.2 == 0.3) << endl;
```

Rather perplexingly, we see that `0.5 + 0.5 == 1.0` and `1.0 + 2.0 == 3.0`, but for some reason, `0.1 + 0.2 != 0.3`. Try playing around with other values or operations and see what happens as a result.

The truth is that the computer doesn't actually make computations with the real number system, just an incredibly useful subset/approximation of it that serves as a shockingly good model for most practical computations. As with any model, though, it has its imperfections, and if handled without care, these imperfections can end up rather massive in scale.

A lot of computational geometry problems inescapably require floating point operations, and ensuring numerical stability is one of the many contributing factors that makes them some of the more difficult problems in a given set.

1.1 Avoid Floating Point Entirely

The first step to minimizing floating point error is to try to avoid using them at all. A lot of problems will give the input entirely in integers, and one of your first instincts should be to try to see if we can *keep* everything in the integers, even if it might be less straightforward.

Example 1.1. Given two fractions $\frac{a}{b}$ and $\frac{c}{d}$, where $-10^9 \leq a, b, c, d \leq 10^9$ are integers, with $b, d \neq 0$, output whether the two are equivalent fractions.

Instead of checking `a/b == c/d`, we can rephrase this problem as checking if `a*d == c*b`, which is logically equivalent to the first statement but crucially avoids using division. Also important to check is that we don't get any integer overflow: in this case, notice that $ad \leq 10^{18}$, so as long as we are using `long long`, we will be safe.

Example 1.2. Given a disc with center $C(h, k)$ and radius r and we are given a point $P(x, y)$, where $-10^9 \leq h, k, r, x, y \leq 10^9$ are all integers, output whether P is contained within the disc.

Geometrically, we see that P will be contained within the disc iff $|PQ| \leq r$. However, $|PQ| = \sqrt{(x-h)^2 + (y-k)^2}$, and that square root operation will introduce a floating point number into our computations! We can avoid that square root simply by squaring both sides; we instead check if $|PQ|^2 \leq r^2$, which will only involve integers that fit within a `long long`.

Exercise 1.3. Is it always true that $a^2 < b^2$ is equivalent to $a < b$? When does it hold true,

and why can we use it here?

Exercise 1.4. Given two circles whose centers are (h_1, k_1) and (h_2, k_2) and whose radii are r_1 and r_2, respectively, where $-10^9 \leq h_i, k_i, r_i \leq 10^9$ are all integers, output whether the two circles are intersecting. If yes, how many points of intersection are there?

Sometimes, even with all integer inputs, a floating point number will be inescapable in the problem due to division, square root, a trigonometric function, or any number of reasons. A good rule of thumb is to try to minimize the number of floating point operations in your program. Stay in the integers for as long as you can, and convert to floating point all at the end if possible. This actually won't always be the case, but it's a helpful heuristic to have, especially when it's just division.

Example 1.5. Given an array of $1 \leq n \leq 10^5$ integers $a_1, a_2, \cdots a_n$, where each element is an integer from -10^9 to 10^9, output the arithmetic mean of the array.

In this example, it would be much more preferable to perform the sum $a_1 + a_2 + \cdots + a_n$, then divide the sum by n, rather than the sum $\frac{a_1}{n} + \frac{a_2}{n} + \cdots + \frac{a_n}{n}$. Even though the two are mathematically equivalent, one is more precise than the other when performed by a computer.

1.2 Epsilon Hack - Margins of Error

One common—and I must point out, *irresponsible*—way to deal with floating point errors is to allow for some margin of error, which we denote mathematically as ε, and in code as `EPS`. Consider the following problem.

Example 1.6. Given a line l of the form $ax + by = c$ and a point $P(m, n)$, where a, b, c, m, n are **not necessarily integers**, output whether point P is on the line.

Bounds are intentionally left vague for now.

Mathematically, one can simply plug the value of the point into the equation for the line and check if the equation $am + bn = c$ is true or not. However, we know that we can't entirely trust equality checks when we are dealing with floating point integers.

One very easy way to attempt to remedy this is that instead of checking if $am+bn$ is exactly equal to c, we check if it's *close enough* to c, at most some epsilon away. In other words, we replace any tests for equality `x == y` with $|x - y| < \varepsilon$, which we implement in C++ as `fabs(x - y) < EPS`. This epsilon amount of "wiggle room" is us saying that if two numbers are close enough, then we can consider them equal anyway.

As a side note, a more accurate way to check if the point P is on l is to compute whether the distance from P to l is less than ε, although the idea of a margin of error is still the same.

This epsilon technique is also helpful in some other scenarios. These include:

- If you are taking the square root of a number that could potentially be `0.0`, keep in the mind the possibility of a "-0.0", or some small, seemingly insignificant negative value instead of 0. Taking the square root of a negative number will result in `NaN`s (Not a Number) that will break your program. You may feel inclined to sprinkle some epsilons around to fix the problem.
- Be careful when using `floor` and `ceil` on doubles. For instance, even if x should equal to

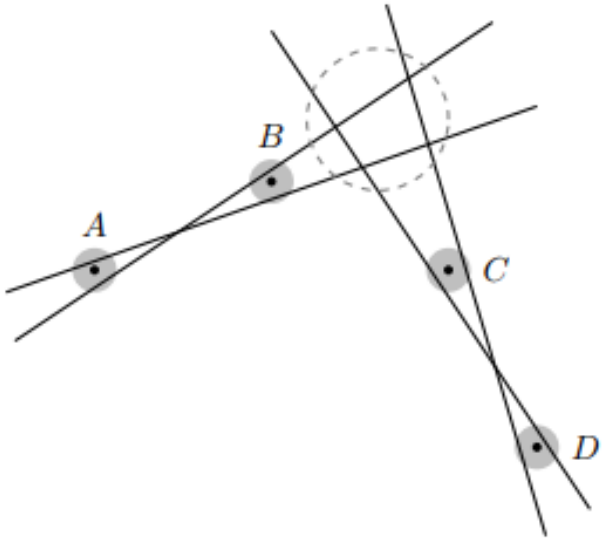
`2.0`, `floor(x)` might still end up evaluating to `1.0` if the value of x is `1.999999999034` or something. Likewise, be careful with ceil. Taking `floor(x+EPS)` helps address this.

The question then becomes—what is an appropriate value for ε? Is 10^{-5} or 10^{-6} enough? Or maybe it has to be 10^{-9} or 10^{-12} to be accurate enough. This is one of the main reasons why sprinkling epsilons haphazardly around your code is typically considered a hack. It is **black magic**. You are not entirely sure about the correctness of your program and are entirely reliant on proof by AC to show that what you did was sufficient. In fact, finding the right ε can sometimes end up being trial and error. That's not even mentioning the fact that ε won't help address other floating point errors, especially the ones that are much larger in scale.

So, while I bring it up as a useful tool for in the middle of a contest and you really have no other options, there are much more proper ways to do things. The "proper" way actually does build on this idea of an epsilon margin of error, except in a far more precise manner.

1.3 Small Imprecisions Can Become Big Imprecisions

Let's interject with a concrete example of things going catastrophically wrong that the epsilon hacks do not help with at all. This example, and much of this section, comes from the first chapter of Lecomte's book on Computational Geometry. Consider finding the point of intersection of two lines, where each line is defined by two points.



Due to the imprecision of floating point numbers (which we will elaborate on later), consider that even though we know where point A is *supposed* to be, what we end up storing could anywhere within that disc around A; the same applies for the other three points.

You can trace (even if just with your finger) where the intersection of AB and CD is *supposed* to be. The disc enclosed by the dotted line, however, represents the set of all possible values that *could* be the center, given the initial imprecision. Notice that even very small circles can scale out to a *huge* range of possible answers, some of which might end up nowhere near the actual answer.[1]

[1] A more physical example of this phenomenon (errors blowing up) is the propagation of error in ruler-and-compass constructions. Here's a good blog post about it by Joel David Hamkins: http://jdh.hamkins.org/propagation-of-error-in-classical-geometry-constructions/

We will see in the next section what exactly can cause this, but realize that small margins of error can blow up catastrophically.

1.4 Precision Guarantees

First, let's identify the two types of floating point error. Suppose that x is the real, pure, mathematically correct answer, and x' is the computed answer. Then,

- $|x - x'|$ is the **absolute** error.
- $\left|1 - \frac{x'}{x}\right|$ is the *relative* error.

As hinted at by their names, absolute error measures the actual numerical value of the error, while the relative error attempts to sort of *scale* itself according to the magnitude of the error. You might be familiar with these terms from programming competitions, which state, for instance "The answer will be correct if the absolute or relative error is less than 10^{-6}", meaning either of the two values stated above.

Suppose that in science class, you were asked to measure the length of your cellphone, so you pull out a standard 30cm rule and receive an answer of 13.8 centimeters. Except, if you recall your lessons on significant figures, it is more accurate to state that the actual length could be anywhere from 13.75 to 13.85, or 13.8 ± 0.05 centimeters. So, we have a measured answer—the value 13.8 which we got from the ruler—but we admit that due to physical limitations, the *true* answer could be anywhere within the given range. We treat floating point numbers similarly.

Firstly, it must be said that even when using **`double`** or **`long double`**, small enough integers are stored exactly, so as long as the answer is within around $\pm 9 \times 10^{15}$ for **`double`** or $\pm 1.8 \times 10^{19}$ for **`long double`**, and you are *only* using $+$, $-$, and $\times$, your answer will still be exact. Note that the **`long double`**'s integer range can do everything that a **`long long`** can.

Now, what about non-integers, or extremely large integers? If the input is exact (meaning either all past operations have been on integers, or this is the very first operation on a floating point input), then any single arithmetic operation of $+$, $-$, $\times$, $\div$, or $\sqrt{x}$ will yield a *relative* error that will not exceed ε. The value of this ε, called the machine epsilon, is guaranteed to be $< 1.2 \times 10^{-16}$ for **`double`** and $< 5.5 \times 10^{-20}$ for **`long double`**.

Now, consider the maximum possible value attained at any point in your calculations; call this value M. Since the relative error is at most ε, the absolute error of values up to size M will be $M\varepsilon$. More generally, if n addition and subtraction operations are performed, then the relative error will be at most $nM\varepsilon$.

Say we wish to get the sum of up to 10^5 numbers who absolute value does not exceed 10^9. Here, M would be 10^{14}, which would be when all values are maxed out. Thus, the absolute error is guaranteed to be at most 1.2×10^{-2} when using a **`double`**, and at most 5.5×10^{-6} when using **`long double`**.

To incorporate multiplication in, Lecomte explains it in terms of geometry. Each value has a *dimension*; for instance, length is a 1D value, area is a 2D value, volume is a 3D value, and so on, with constant factors like π being a "0D" value. Usually when we multiply an a-dimensional value with a b-dimensional value, we get an $a + b$ dimensional value.

Input is usually as coordinates or some other 1D value, and say we find a largest value M that bounds them. Again, since we get higher-dimensional values by multiplying smaller dimensions together (multiplying length by length gives area, for example), then 2D values are bounded by M^2, 3D values are bounded by M^3, and etc.

Suppose the highest dimension attained in our calculations is d; then, the maximum absolute error is approximately $nM^d\varepsilon$. Note that this means that repeated multiplication on doubles can get extremely inaccurate rather fast, as that M^d blows up in magnitude.

Note that although this bounds the absolute error of multiplication, the *relative* error is actually preserved quite well. Intuitively, this should make sense when you consider that doubles are basically stored with "scientific notation" style, which consists of a mantissa containing significant digits multiplied by an exponent. Recall that multiplying with scientific notation essentially boils down to multiplying the significant digits and adding the exponents together.

1.5 Things Going Wrong: Subtraction

Subtraction isn't entirely perfect, however. Try the following—let $a = 10^9$ and $b = 10^{36}$, stored in floating point types. Now, compare the outputs of `(a + 1) - a` and `(b + 1) - b`. Mathematically, these should both somewhat obviously give an answer of 1—except only the former gives an answer of `1.0`, while the latter gives an answer of `0.0`!

Recall that a is small enough that we can do exact arithmetic on it, as if it were stored in an integer data type. However, b is large enough that we have to use the "scientific notation" style of storage that floating point numbers use. And, at the scale of 10^{36}, adding a $+1$ is nowhere near significant. So, $(b + 1)$ gets stored as being pretty much the same as b; that $+1$ just magically vanishes, and it becomes apparent when we subtract the two.

This is one reason why Lecomte argues that the "absolute or relative error" system is imperfect. Unfortunately, this is the best system we have for contests at the moment, so you should just be aware of this and try to be careful with it. My advice is that when adding two numbers of opposite signs, try to do some mathematical manipulations to keep the operands relative close to each other in terms of magnitude.

1.6 Things Going Wrong: Division and Square Roots

Here are the last two operations, which are often where lots of big errors from small imprecisions have come from. If your inputs are integers, then based on the precision guarantees from earlier, you should feel reasonably safe with these operations, so long as you take care not to fall to catastrophic cancellation.

However, on general floating point numbers, things can get messy. Intuitively, if we add two numbers whose margin of error is $\pm 10^3$, then their sum is at most $\pm 2 \times 10^3$ from the actual answer, which doesn't look too bad. However, division and square roots don't behave as nicely. Note that if x is close to 0, $\frac{1}{x}$ actually ends up blowing up to some enormous value! Although not as dramatic, square roots also end up making x *bigger* when $0 < x < 1$; note that $\frac{1}{4} < \sqrt{\frac{1}{4}}$, for instance. Note that a margin of error of 10^{-8} becomes a margin of error of 10^{-4}, a whopping 4 orders of magnitude higher (and this only gets worse as x approaches 0).

The only real tip I have is to be careful when dealing with small numbers near 0. In general—so this also includes other real-valued functions like the trigonometric functions, logarithms, and etc—just try to minimize the number of "dangerous" items in your code.

Note that these only behave poorly with imprecise inputs, such as intermediate values that have already gone through some computations; once again, exact inputs will still give a relatively small margin of error.

1.7 Problem Conditioning

Sometimes, the problem itself is inherently ill-defined. Consider the following two problems:

Problem 1.1. Given a point $P = (x_p, y_p)$ and a line determined by two points $A = (x_a, y_a)$ and $B = (x_b, y_b)$, determine if P is located counterclockwise of B from A, or clockwise of B from A, or on the line AB.

Problem 1.2. Given a point $P = (x_p, y_p)$ and a line determined by two points $A = (x_a, y_a)$ and $B = (x_b, y_b)$, determine the *signed distance* from point P to line AB. The signed distance is like the distance, except its sign encodes where P is located with respect to the line AB: it's negative if P is clockwise of B from A, positive if it's counterclockwise, and 0 if P is on the line.

These problems are clearly related. In fact, you can solve Problem 1.1 using Problem 1.2 by simply using the sign of the signed distance. However, we shall see in a bit that they differ in an important way *numerically*.

These two problems are often used as subroutines as part of bigger algorithms in computational geometry. They have simple mathematical solutions using some tools from vector algebra, but if you don't know that yet, don't worry. We'll proceed below using only the more elementary stuff.

Specifically, we'll solve a special case of Problem 1.1 where *P and B are to the right of A*. (Any other case can be reduced to this by 90-degree rotations and/or reflections about the axes.) So we will assume that $x_a < x_p$ and $x_a < x_b$. In this case, all we need to do is compare the *slopes* of the lines AB and AP. Check that:

- If $\mathit{slope}(AB) < \mathit{slope}(AP)$, then P is counterclockwise of B.
- If $\mathit{slope}(AB) > \mathit{slope}(AP)$, then P is clockwise of B.
- If $\mathit{slope}(AB) = \mathit{slope}(AP)$, then P is on the line AB.

In other words, we only need the sign of $\mathit{slope}(AB) - \mathit{slope}(AP)$. Also, the slope of the line through points (x_1, y_1) and (x_2, y_2) is simply $\frac{y_2 - y_1}{x_2 - x_1}$, so all in all, the algorithm amounts to checking the sign of

$$\frac{y_b - y_a}{x_b - x_a} - \frac{y_p - y_a}{x_p - x_a}.$$

Now, from what you've just learned, fractions force us to use floating point numbers which can be nasty, so we'd like to stay in integers as much as possible. Luckily, both denominators are positive (why?), so we can multiply by $(x_b - x_a)(x_p - x_a)$ without changing the sign! So the sign is the same as the sign of

$$(y_b - y_a)(x_p - x_a) - (y_p - y_a)(x_b - x_a). \tag{1}$$

If the coordinates are integers, then this is also an integer, so we're perfectly good since we stay in integers.

On the other hand, if the coordinates are floating point numbers, then this will be a floating point number as well, and we now have to worry about precision errors. But this expression is pretty bad! It has lots of subtractions, which as we've seen earlier is pretty bad—it magnifies errors so much, that even sprinkling your code with lots of ε may not work. There may not even be a choice of ε that works for all cases because of all these catastrophic cancellations we get from subtraction!

There are other possible problems. For example, if you derived the solution in a different way, then you may end up with a different expression like the following:

$$(y_b - y_a)(x_p - x_b) - (y_p - y_b)(x_b - x_a). \tag{2}$$

Exercise 1.7. Show that formulas (1) and (2) always have the same sign.

So these formulas have the same sign...at least mathematically. However, with floating-point numbers, all bets are off! You certainly can find cases where they differ in signs. (See problem **C1**.)

Taking a step back, we see that there's a real issue inherent with this problem: it is *ill-conditioned*! The main issue is that it's very sensitive to perturbations of the input: a small change in the input may result in a large change in the answer. Indeed, although (1) is continuous in the input variables, its *sign* is not, so even if we vary the input *continuously*, there will be huge, sudden jumps in the output. For example, suppose P is on the line AB but we don't precisely know the coordinates to full precision, then we can never really be sure that P is actually on the line!

Compare this with Problem 1.2. There, if you try moving the points P, A and B around, then you can see that the distance from P to AB changes continuously as well. We can see this even without deriving the solution to the problem—in other words, using just our imagination! In particular, small changes in the input lead to small changes in the output. (The output itself may switch signs abruptly, but that's okay since we don't need the sign.) So we say that the problem is *well-conditioned*.

So now we see that there's a crucial difference between the two problems, which is that one problem is better-conditioned than the other. Conditioning is an inherent property of the *problems* themselves, not of the algorithm used to solve them.

Another example of functions with discrete jumps is when computing the floor or ceiling of a floating point number. If your number is very close to an integer, then varying it by small amounts can drastically change the output. So it's ill-conditioned as well.

Note that discrete jumps aren't the only way for something to be ill-conditioned. Even if the output depends continuously on the inputs, it may still be that small changes in the input lead to large changes in the output.[2] We've seen this with the geometry example above, where the small errors get magnified.

Of course, algorithms can behave badly in a similar way too—small change in input may lead to drastic changes in behavior—but if it's the algorithm we're talking about, then we usually use different jargon: we say the algorithm is *numerically unstable* or *stable*.

How does it affect you, as a progrmamer? Here are a few possible takeaways:

- Of course, awareness is key. Be aware that numerical stability and conditioning exists. In general, try to think about the numerical stability of what you're coding. It's not a good idea to be gung-ho and use floats everywhere without thinking!
- If you're using a known algorithm as a subroutine, you'll want to know how numerically stable they are. Some are better than others. And sometimes, there's a tradeoff between numerical stability and efficiency.

[2] This happens in general if the "*derivative is large*." I put it in quotes since we haven't defined what the derivative is yet (much less for functions with more than one input variable), and even what "large" is. In numerical analysis, they usually quantify the extent to which a function is ill-conditioned, and this measure of ill-conditioning is usually called the condition number. You don't have to know what this is yet, just that it exists, though feel free to skim the Wikipedia article to get a feel for it.

As a fun fact, it turns out that the formula (1) actually works generally! In other words, we can drop the assumption that B and P are to the right of A, and the answer to Problem 1.1 depends on the sign of (1) in exactly the same way. You can check this by tedious case-by-case analysis—by checking all possible combinations of locations of B and P with respect to A—or you can prove it more elegantly using basic knowledge of vector algebra (which you'll learn later once you learn computational geometry), by noticing that it's just the cross product of $P - A$ and $B - A$. (The quantity $x_1y_2 - x_2y_1$ is called the 2D cross product of vectors $\langle x_1, y_1 \rangle$ and $\langle x_2, y_2 \rangle$, and it's a very important quantity you'll learn about later.)

2 Randomized Algorithms

Many classic algorithms that we've learned are **deterministic**, meaning that, given the same input, the algorithm will always produce the same output, and the series of steps which it executes will always be the same. In other words, the input totally *determines* how the algorithm performs.

Consider an algorithm like selection sort. Roughly, the algorithm goes: locate the smallest element in the array, move it to the front, then recursively sort the remaining array. Although the algorithm will behave differently given different arrays as input, we know that every time we run it on the array `3 1 4 1 5 9`, it will always execute the exact same steps, whether we do it today, tomorrow, or two years from now.

Randomized algorithms, on the other hand, as the name suggests, involve some random elements in their execution. Thus, even if we run the same program on the same input, the program might behave differently from one execution to the next. As convention, whenever I say 'random' in this module, assume it to mean 'with uniform distribution' unless explicitly stated otherwise.

2.1 Random generation

Before using randomization in algorithms, let's first discuss a bit how a computer generates random numbers in the first place.

It turns out that it's hard to get *truly random* numbers for use in your program. To start with, where do you get them? Maybe from the system clock? The temperature of your computer? Maybe your computer has a dedicated machine solely for generating random numbers? (Though you can still ask how that machine get its random numbers.)

There are a few problems with these. First, they may not be reliable: maybe your system clock only stores millisecond-precision time. Or maybe your program checks the clock very regularly (say in a simple for loop) so you're getting numbers in roughly an arithmetic sequence (not random!). Or maybe reading the clock so often is slow, since your program has to communicate with the OS often. Or maybe, checking the temperature of your computer is impossible, or simply disallowed in the judge machine!

So to work around these, programs instead use so-called *pseudorandom number generators*, or PRNGs. A PRNG usually takes in a number called a *seed*, and produces a sequence of "random-looking" numbers. The same seed always produces the same sequence of numbers.

With a PRNG, we don't have to worry about generating many random numbers anymore! All we need is to generate a random *seed*, which is typically small (e.g., 64 bits), so usually the computer time (in nanoseconds) is enough, so we only need to do it once (at the beginning). PRNGs usually generate numbers pretty fast, and although they're technically predictable, they're usually *hard* to predict, especially without knowing the exact seed. This makes it suitable for randomized algorithms.

Two popular PRNGs are the linear congruential generator (LCG), and the Blum Blum Shub PRNG. Each of them takes the form $x_{i+1} \leftarrow f(x_i)$ for some function f, that is, they take the seed value and iteratively apply f to produce the sequence.

- For LCGs, $f(x) = (ax + b) \bmod m$ for fixed numbers a, b and m (and m is often chosen to be prime).
- For Blum Blum Shub, $f(x) = x^2 \bmod m$ where m is the product of two distinct primes.

As you can see, PRNGs don't really generate random numbers. However, they generate numbers that are "random enough" for most purposes. But how does one check that a sequence of not-really-random numbers is "random enough"? It's a tricky business, but in general, you'll want it to pass some statistical tests that a true random sequence passes. For example, you'll want to check that it's not periodic (or at least the period is long enough), it doesn't have biases or other sorts of regularities, etc.[3]

Using such tests, we can determine which PRNGs are good for use in algorithms, and which are not. For example, LCGs are known to have a few flaws and fail certain statistical tests. On the other hand, the Blum Blum Shub generator is known to be good, as long as you only take the last few bits as your "random" sequence, and also if the prime factors of m are chosen well.

In C++, the (bad!) function `rand()` uses a bad LCG, so don't use it! Instead, use `mt19937`, which is based on a different PRNG called the **Mersenne Twister**, which is far better.

Notes:

- Don't hardcode a fixed seed! This makes your solution hackable, since anyone who sees it knows the exact sequence of "random" numbers that your program will use, making it deterministic. Instead, use the system time.
- Actually, it's very tricky to obtain *truly random* numbers, not just in programs, but even in principle! Thinking about this leads you to the question "What is randomness, really?" This is a question best left to the philosophers.

2.2 Do Not Use rand()

This Codeforces blog post asserts that you should **not** use the naive `rand()` function in C++, and thus also other methods that rely on it such as `random_shuffle`. The gist of it is that the maximum value `RAND_MAX` is only guaranteed to be at least $2^{15} - 1$, or 32767. That's tiny! If you only generate random numbers from 0 to 32767, then for instance, you will be biased towards only selecting the first few items from a large array, or will not end up scattering the elements in an array that much using `random_shuffle`.

The post suggests the use of the `chrono` and `random` libraries included in C++11 and later. For all intents and purposes, whenever you need a random integer, you can blackbox-use the following code. Create an `rng` (random number generator) with the following code,

```
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
```

Of course you can name the random-number-generator anything you want, though I always just use rng, and you don't need more than one anyway. Here, the `mt` stands for Mersenne Twister, a pseudo-random-number generator which has the period length of a Mersenne Prime.

Then, to generate a random integer in the inclusive range $[a, b]$, we use the expression,

```
uniform_int_distribution<int>(a, b)(rng)
```

passing the `rng` from earlier as a parameter.

To randomly shuffle around the elements of an array or vector, we use `shuffle`, passing

[3] A popular collection of statistical tests are the diehard tests, created by George Marsaglia in 1995.

a reference to the start and end of the to-be-shuffled list, then including the random-number-generator we will be using. So, for example, we could write

```
shuffle(a.begin(), a.end(), rng); // if a is a vector
shuffle(b, b+n, rng); // if b is an array
```

It's a bit more of a mouthful than just `srand(time(NULL));` and `rand()`, but it's important to get used to it to properly make random data.

2.3 Randomized Quicksort

Now, let us consider first the classical example, which is the randomized version of Quicksort.

Suppose we had some deterministic method of choosing our pivot in Quicksort; say, for example, you always chose the first element as your pivot. It was asserted that Quicksort is $\mathcal{O}(n \lg n)$ in the average case analysis, but is technically $\mathcal{O}(n^2)$ in the worst case analysis. Although for most input, the sorting algorithm is efficient, there exist some degenerate cases where the algorithm devolves to $\mathcal{O}(n^2)$.

Exercise 2.1. Construct an array that takes $\mathcal{O}(n^2)$ comparisons to sort using a Quicksort that always chooses the first element as its pivot, then places the other elements into two separate lists (less than or equal to the pivot, or greater than the pivot), with order preserved.

Even if we can show that these degenerate cases are few and far between, a determined problem setter (and all the hackers on Codeforces) will go out of their way to explicitly construct these degenerate cases to cause your program to TLE.

One benefit that randomized algorithms have in general is that they can 'force' the average "random" case analysis. Randomized Quicksort is still $\mathcal{O}(n^2)$ in the worst case, but that shouldn't happen often, and there is no way for a malicious party to force that worst case to happen with a pre-written test case.

So, suppose we choose a random element as our pivot instead. What is the expected value of the number of comparisons needed to sort an array of n elements?

First thing we can do — experiment! You have a computer, so why not use it? Proving the efficiency of a randomized algorithm can be tricky, so in the middle of a contest, it may be worth it to simply write the algorithm and test it locally on your machine. Is it fast? Keep track of how long it takes or how many operations it's making. Maybe that's enough to convince you that this algorithm might just be fast enough.

Of course, if you are the problem setter—or a mathematician—then you won't be satisfied with this experimental data. We'd like to really prove that randomized Quicksort is efficient. Here is a simple and hopefully intuitive proof that the randomized Quicksort is $\mathcal{O}(n \lg n)$.

Let $e_1, e_2, \cdots e_n$ be the elements of a, except already sorted in ascending order. So, e_1 is the smallest element of a, e_2 is the second smallest element, and so on, until $e_n$is the largest element of a. For now, we assume that all elements of a are distinct. If there are duplicate elements, then the algorithm will only improve anyway, so in establishing an upper bound, it is sufficient to only consider the distinct case.[4]

[4] Actually, this is not necessarily true! For example, if you always place elements equal to the pivot into one of the buckets (say the lower bucket), then there's some bias that the lower bucket is larger. To fix this, put the equal elements into a third bucket.

Now, let's choose a random pivot. The pivot divides the array into two buckets, and the respective sizes of the buckets depends on the pivot's place in the sorted array e; everyone less than the pivot ends up in the left bucket, and everyone greater than the pivot ends up in the right bucket. If the chosen pivot is e_i, then there are $i-1$ elements in the left bucket and $n-i$ elements in the right bucket.

We can roughly understand what happens in the extreme cases. If we pick a pivot close to the middle, and thus the buckets are roughly the same size, then we get the $\mathcal{O}(n \lg n)$ time due to Divide and Conquer. But, if we pick a pivot close to the edge, then we devolve into $\mathcal{O}(n^2)$ since it sort of acts like Selection Sort or Bubble Sort, with extremely lopsided bucket sizes.

Now, consider the "middle" part of e. Divide e into four contiguous segments that are roughly the same size. We will consider two cases here—if the chosen element is in the 1st or 4th segments, the "outer" segments, or if it is in the 2nd or 3rd segments, the "inner" segments. If the pivot is in the inner segments, then it is "good", and if it is in the outer segments, then it is "bad". We see that if the pivot is randomly chosen, it is good with probability $\frac{1}{2}$. It turns out that even if the pivot is good only about half the time, that is already sufficient to show that it is $\mathcal{O}(n \lg n)$.

For each case, we simply assume the worst possible scenario. For the good pivot, we assume that the split is in a $1:3$ ratio, and for the bad pivot, we assume that we picked the max or min so we have a bucket with 0 elements and a bucket with $n-1$ elements. This would give the recurrence,

$$T(n) = \frac{T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right)}{2} + \frac{0 + T(n-1)}{2} + \mathcal{O}(n)$$

where $T(n)$ is the *expected* runtime of the algorithm. You can pick your favorite recurrence-solver to show that this $T(n)$ is $\mathcal{O}(n \log n)$. The essence is that whether we get a good pivot or a bad pivot is a coin toss. It's unlikely to win *every* coin toss, but similarly it is unlikely to *lose* every coin toss. As long as we win *enough* of the coin tosses, our algorithm performs efficiently.

Exercise 2.2. Show that if $t(0) = 2$ and

$$t(n) = \frac{t\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + t\left(\left\lfloor \frac{3n}{4} \right\rfloor\right)}{2} + \frac{0 + t(n-1)}{2} + n$$

for all $n \geq 1$, then $t(n) \leq 4n \lg n + 2n + 2$ for all $n \geq 1$.

This illustrates why $T(n) = \mathcal{O}(n \log n)$ in the recurrence for randomized quicksort.

Hint: Use induction. Be careful with your base cases!

2.4 On Malicious Test Data

Hold on, so what's wrong with just taking the first element as our pivot each time? It's not like the first element is "more or less random" than the third, or seventh, or any other element. And that's true!

The thing is, in truly random data, we said that it would be an incredibly contrived set of circumstances that would lead to us getting our $\mathcal{O}(n^2)$. But, if we did something as predictable as always picking the first element as our pivot, then bad test data can explicitly be constructed that "forces" a TLE. There is a laughably minuscule chance of randomly stumbling upon the "bad" cases—but if you're too predictable, a hacker or evil problem setter can set the scenario just right such that getting the TLE isn't so "random", but premeditated.

Imagine you are playing a series of n games of Rock-Paper-Scissors with someone and you need to win all the fights. If they are playing truly randomly and unpredictably, then the best you can do is guess at every game, giving you a success rate of only $\frac{1}{3^n}$ to win every game. But, if you begin to notice patterns like "Always plays rock", "Rotates through Rock-Paper-Scissors", or "Play what beats the opponent's previous move", then you can come up with a better-than-random strategy to beat them consistently.

For this reason, if a Codeforces problem needs sorting, you need to be careful when using Java's `Arrays.sort()`, which is a non-randomized Quicksort! You will surely end up getting hacked. Read this link for details, but basically, randomly shuffling the array at the start of the algorithm serves the same purpose as choosing a random pivot at each step, which is an easy enough fix. Alternatively, note that randomly shuffling an array is basically equivalent to sampling from all possible orderings of an array, which turns the *average* complexity ($\mathcal{O}(n \log n)$) into the *expected* complexity.

That's what randomized Quicksort, and many other randomized algorithms, do. They "inject" the randomness into your algorithm, so that it's almost impossible to find the magic case which breaks your solution, and any explicitly constructed malicious test data gets reduced to random noise as well.

2.5 On Probability

You might still have some reservations about randomized algorithms, so let's talk a little bit about statistics and probability.

Let's consider a randomized algorithm that has a *one-sided* error. For example, consider a problem with a yes-no answer such as: *Given n, is n prime or not?* Suppose you have an algorithm that:

- if n is prime, always returns **yes**.
- if n is not prime, returns **no** with probability 50%, and **yes** otherwise.

The one-sidedness comes from the fact that it only goes wrong in one of the cases (the **no** case).

You can think of such an algorithm as relying on a coin toss (if n is not prime): if the coin lands on heads, then the algorithm is correct; tails, incorrect. This algorithm is thus wrong 50% of the time, which makes for a pretty terrible algorithm.

However, what if we run the algorithm *twice*? If it says **no** in either run, then we know the answer is **no** (since we never say **no** when the answer is **yes**). In terms of coins, the algorithm is now only incorrect if the coin lands tails on both of them. Now the algorithm is wrong $\frac{1}{4}$ of the time, which is 25% of the time. Better, but not out of the realm of possibility.

However, let's jump to 10 runs! The algorithm is only incorrect if the coin lands on tails on all of them. This doesn't sound like too much—except now the chance of being wrong is $\frac{1}{2^{10}}$, which is less than 0.1% chance to be wrong. If we do 64 coin tosses, then $\frac{1}{2^{64}}$ is approximately 10^{-19}; it is about five orders of magnitude *less* likely than winning the lotto twice in a row! My coach has even said that it is more likely for the judge servers to spontaneously catch fire than it is for an algorithm with those chances to fail.

Suppose there is a staggering 500 test cases! What is the probability that a program that fails $\frac{1}{2^{64}}$ of the time will fail at least one of them? We get $1 - \left(1 - \frac{1}{2^{64}}\right)^{500}$, and plugging that into Wolfram Alpha yields... 2.7×10^{-17}.[5] Still insignificant, so we're still almost guaranteed to succeed!

[5]Some useful approximations if you don't have Wolfram Alpha on hand: $(1+x)^r \approx 1+rx$ when $|rx| \ll 1$ and

This is one way that random algorithms ensure correctness. Although there *is* a chance that the algorithm fails, we have the power to make that error percentage as arbitrarily small as we want it to be. In many cases, all you have to do is run the algorithm many times until you get the desired probability bound. (Note that all our probability calculations above are just lower bounds, since it assumes that all answers are **no**, which is probably not true! If some answers are **yes**, then we're right on them 100% of the time.)

Miller-Rabin's test for primes is one such randomized algorithm with one-sided error that employs this principle. Given a number n, if it's prime, Miller-Rabin determines it with 100% probability, and if it's not prime, then it determines it with *at least* 75% probability. (The actual probability depends on n; the probability is much higher for some n than others. 75% is just a proven guaranteed lower bound.)

The other way is more similar to our analysis for randomized Quicksort earlier. Suppose the efficiency of part of your program relies on some random element; as the random element is called more and more times, there is a tendency for the average over all trials to tend towards the expected value. In other words, sometimes you get heavy degenerate cases, but those are rare, and balanced out by the equally likely "very nice" cases. So, even though there is a chance that individual actions are expensive, a sort of amortization keeps all the actions together as generally efficient.

Finally, we can leverage expected values in another way. Suppose you have a program that has a $\frac{1}{n}$ chance of working, and it takes $\mathcal{O}(n)$ time to evaluate whether a certain random value is a solution or not. Since it is a $\frac{1}{n}$ chance for success, it is expected that it will take around n tries until your first success. So, if we just kept trying truly random values until we got a correct answer, we could expect our program to usually take $\mathcal{O}(n^2)$ time.

Exercise 2.3. Show that if the chance of success is p, then the expected number of tries to succeed is $1/p$.

Hint: You'll need an infinite sum. Compute the probability that you succeed on the kth try, and use the linearity of expectation.

2.6 Confidence with Random Experiments

If the problem were as easy to analyse as tossing coins, then it's also easy to be mathematically sure of its accuracy. However, what if the randomness is a bit more complicated to analyse?

Example 2.4. Produce any 500 points in $[0, 999] \times [0, 999]$ such that no three of them are collinear.

This has many solutions, but let's look at a randomized one. Push all 1000^2 points into a large array and randomly shuffle them around (read the next section on how to do this *properly*). Then, greedily consider each point—if it is not collinear with any other two points that you have already taken, then take it as well, otherwise ignore it and check the next point. Keep doing this until you have 500 points.

$(1+x)^r \approx e^{\frac{rx}{x}}$ when $|x| \ll 1$

Exercise 2.5. Look at the step where we consider a candidate point and check if it is collinear with any other two points that are already taken. Naively, this can be done in $\mathcal{O}(n^2)$ by simply doing a collinearity check with every pair of points that was already taken (and $\binom{n}{2} = \mathcal{O}(n^2)$).

Find a way to perform this check in $\mathcal{O}(n \lg n)$, where n is the number of points already taken.

So, first we spend $\mathcal{O}(N^2)$ time to random shuffle around the points, where N is the maximum value of one of the coordinates of a point, which is 1000 in this problem. Then, the complexity of the greedy part is

$$\mathcal{O}(n \lg n \times \text{Expected number of points we need to check before taking } n \text{ points}).$$

Here, we can let n be the number of points we need to take, which is 500 in this problem.

But what *is* the expected number of points we need to check before we get a set of 500 non-collinear points? The analysis for this... looks incredibly hassle. But, we have a computer, so what can we can do instead is run experimental data!

First we run the program a few times locally, and add an extra counter that keeps track of how many points we've considered before the program finishes. I ran this a couple of times and got 825, 824, and 818, respectively. That's around the order of n, so it seems our solution is $\mathcal{O}(n^2 \lg n)$, so that passes![6]

If you want to be even more convinced, you have the luxury of running experiments on your local machine. Run this test 1000 times and get the mean and standard deviation. I did this and got a mean of 822.835 with a standard deviation of ≈ 29.3098; if this were really normally distributed, then getting a value more than three standard deviations above would already be phenomenally rare, and that wouldn't even put us over 1000 points!

As long as there is no malicious test data, i.e. explicitly constructed evil cases from the problem setter, then you can reasonably expect other random test data to behave much the same way.

2.7 The Central Limit Theorem

The Central Limit Theorem is one of the most important theorems in statistics, and it allows us to feel safe that many of our different randomized algorithms are correct. In essence, it says that suppose we have a random experiment that produces a random number X every time it is conducted. It is important that each of these trials is independent from one another, e.g. a coin toss is not influenced by past tosses, nor does it influence any future flips. The Central Limit Theorem states that no matter what the experiment is or how its values are distributed, the average value of these Xs over a large enough number of trials will tend towards a normal distribution.

The normal distribution, as a reminder, follows a mostly bell-shaped curve centered around the mean. The curve can be interpreted as a function which tells you the probability that the result of the experiment is a certain value x. Normal distributions are peaked at their mean, which means a value somewhere within that region is going to be the most likely result. Meanwhile, values on the outskirts very far from the mean, although they technically have a non-zero probability of occurring, are so unlikely to ever happen.

If you had statistics in High School, you might possibly remember a lookup table of values that would tell you the probability that a random value would fall some number of standard

[6] By the way, this is very sloppy, since it's clear that the expected number doesn't scale linearly with respect to n. Indeed, when n is the maximum number you can take, then the expected number for $n+1$ will be infinity!

deviations from the mean. The 68-95-99.7 rule tells us that 68% of data lies within one standard deviation of the mean, 95% of the data lies within two standard deviations of the mean, and 99.7% of the data lies within three standard deviations of the mean. It is *incredibly* likely for your results to be close to the mean.

So, we can conduct a statistical experiment. The first step is gathering data; in my experiment in the previous section, I ran the trial 1000 times to get 1000 data points. These 1000 points are only a sample—to have complete data, we would have to run an infinite amount of trials, but it's more than enough for us to get some good data.

Then, we compute the experimental mean μ of all the values, then the experimental standard deviation σ.[7] The Central Limit Theorem tells us that we can expect these values to be normally distributed, so we can use the standard deviation that we found to get the most likely results of our experiment.

We know that 99.7% of the time, a random normal experiment would give an answer 3 standard deviations within the mean. So, in the experiment I conducted with a mean of $\mu = 822.835$ and standard deviation of 29.3098, an overwhelming majority of the time, our random experiment would take no more than around 911 points in the worst case—this is really good!

The main thing to take away from this is that **randomized algorithms are okay**! You just need to be confident in your analysis that the worst case will very rarely happen. Some random algorithms are even effectively as good as deterministic algorithms, especially if we can make the probability of failure arbitrarily small. As my coach told me before, a probability of 10^{-18} for failure is so ludicrously tiny that it is more likely that the judge's servers spontaneously catch fire than it is for your algorithm to randomly fail [citation needed].

2.8 Adaptive Judges

One instance where randomized algorithms may fail is in some interactive problems, especially if the problem says that the judge is "adaptive". An **adaptive** judge is one whose behavior depends on the program it's interacting with.

Let's illustrate this with the following simple interactive problem:

> **Problem 2.1.** Let A be an array of $2n$ bits, n of which are 0 and n of which are 1. This array is hidden from you, but you may ask the judge for the ith bit and it will give $A[i]$ to you. Your goal is to find two consecutive locations i and $i+1$ such that $A[i] \neq A[i+1]$.
>
> Do it in as few queries as possible! Your score depends on the worst-case number of queries across all interactions. The fewer, the better.

I encourage you to try solving it. The discussion below will be more educational if you do.

First, notice that if your algorithm is deterministic, then you're forced to make at least $n-1$ queries. This is because after only $n-2$ queries, you could have gotten a 0 each time, so there are two other zeroes on the array you haven't uncovered, and you don't have enough information to deduce the answer: if you decide by then that $A[i] \neq A[i+1]$, then you may be incorrect, because those locations could have still been zeroes!

[7] recall that $\sigma^2 = \dfrac{\Sigma(x-\mu)}{n-1}$; the denominator is $n-1$ as a sort of error-correcting factor to account for the fact that we are only taking a sample.

Exercise 2.6. More strongly, show that any algorithm actually requires at least n queries to be absolutely sure it's correct, for $n \geq 2$.

What happens when $n = 1$?

Now, as it turns out, there's a pretty nice randomized algorithm for this! It runs in two steps:

1. Find any two locations ℓ and r (not necessarily consecutive) such that $A[\ell] \neq A[r]$.
2. In $\leq \lg n$ steps, find two locations i and $i+1$ between ℓ and r such that $A[i] \neq A[i+1]$.

The second step can be done without the use of randomization.

Exercise 2.7. Show how to do step 2 without randomization.

Hint: What does "$\lg n$" remind you of?

What about the first step? Well, we know that half of the bits are 0, and half are 1, so we can just try random indices! We're guaranteed to get a 0 with $1/2$ probability, so the expected number of random trials to get a 0 is $1/(1/2) = 2$. (See Exercise 2.3.) Similarly, we expect 2 random trials to get a 1. Therefore, we only expect 4 random trials to get two locations with different values!

Exercise 2.8. Show that we actually only expect 3 random trials.

All in all, we now have a randomized algorithm that runs in $\lg n + \mathcal{O}(1)$ expected queries.

However, *what if the judge is adaptive*? From the perspective of your program, an adaptive judge is the same as a regular, nonadaptive judge, in that it answers the queries as any valid judge would. In addition, the answers it gives are *consistent*, that is, there exists a valid array A such that the answers could have come from a nonadaptive judge with array A.

The main difference is that an adaptive judge doesn't fix the array A beforehand! For example, it may have multiple possible As in mind, possibly *all* of them. Some of your queries rule out some possibilities for A, but the important thing is that the judge ensures that its answers are consistent.

In addition, an adaptive judge can be **malicious**! A smart adaptive judge may cleverly choose its answers so that it triggers the worst case of your solution. In particular, it can even negate the advantages of randomization!

Exercise 2.9. Show that an adaptive judge can always *force* the worst case of n queries for $n \geq 2$ for any solution, randomized or not.

So the takeaway here is: **beware of adaptive judges**! You need to approach them differently, and **think of the worst case rather than the expected case**. Of course, you may still use randomization in some parts of your code if the adaptive judge has no control over them, or if you have a proof that the judge will not be able to defeat the randomization.

3 Problems

Solve as many as you can! Ask me if anything is unclear.[8] In general, the harder problems will be worth more points, although I won't be saying which ones are harder.

3.1 Non-coding problems

No need to be overly formal in your answers; as long as you're able to convince me, it's fine!

N1 Given an equation of the form $ax^2 + bx + c = 0$, where $|a|, |b|, |c| \leq 10^{10}$ are integers with $a \neq 0$ and $\Delta = b^2 - 4ac > 0$, print out the two roots with absolute or relative error less than or equal to 10^{-8}. You'll notice the bounds for this problem are incredibly strict!

(a) [2★] What could go wrong with just using $x = \dfrac{-b \pm \sqrt{\Delta}}{2a}$?

(b) [1★] Prove the quadratic formula's "evil twin", which is that $x = \dfrac{2c}{-b \mp \sqrt{\Delta}}$

(c) [1★] Does using only the "evil twin" solve the problems that we had with the original quadratic formula?

(d) [5★] Propose a method of using **both** the quadratic formula *and* its "evil twin" to avoid the floating point error issue.

N2 [3★] Explain how $\sqrt{a^2 - b^2}$ can differ from $\sqrt{a - b} \times \sqrt{a + b}$ in floating point arithmetic.

N3 [4★] Suppose you are given n floating-point nonnegative integers $a_1, a_2, \cdots, a_n$, which are all initially exact. Your task is to compute their sum, $a_1 + a_2 + a_3 + \cdots + a_n$. Is it possible to get a margin of error better than $(n-1)\varepsilon$?

N4 [4★] Explain the issues with the following problem:

> *Given n integers $x_1, \ldots, x_n$ and an integer b, find the smallest k such that $\sqrt{x_1} + \ldots + \sqrt{x_k} \geq b$, or determine if none exists.*

N5 Do Exercise 2.2, and using it, prove that the expected running time of randomized Quicksort is $\mathcal{O}(n \lg n)$.

N6 Here is another formal mathematical proof that randomized Quicksort takes $\mathcal{O}(n \lg n)$ comparisons on average, using a small amount of probability theory.

(a) [2★] First, a fact about probability. Suppose we are rolling a fair n-sided die labeled from 1 to n. Suppose there is some integer $1 \leq m < n$. We roll the die, and if the result is in $[1, m]$, we accept the result, and if it is in $[m+1, n]$, we reroll the die, and we keep rerolling until we get an integer in the range $[1, m]$. Show that this entire process is equivalent to rolling an m-sided die.

Let $e_1, e_2, \cdots e_n$ be the elements of a, except already sorted in ascending order. So, e_1 is the smallest element of a, e_2 is the second smallest element, and so on, until e_n is the largest element of a.

[8] Especially for ambiguities! Otherwise, you might risk getting fewer points even if you *technically* answered the question correctly.

(b) [2★] Assume that all the elements of a, and thus of e, are distinct. Why is it okay for us to assume this?

Let $X_{i,j}$ be a random variable that is equal to 1 if e_i is compared with e_j, and 0 otherwise. To get the expected number of comparisons, we have to first answer what the probability is that e_i is compared with e_j. We are now looking for the sum

$$\mathrm{E}\left[\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{i,j}\right]$$

and due to the linearity of expectation, this is

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n} \mathrm{E}[X_{i,j}].$$

Now, let's recall some facts about how Quicksort is executed. Describe what happens to the value of $X_{i,j}$ for each of the following scenarios,

(c) [1★] Either e_i or e_j is the randomly chosen pivot.

(d) [2★] The chosen pivot lies between e_i and e_j.

(e) [2★] The chosen pivot is less than e_i or greater than e_j.

(f) [3★] Now, find $\mathrm{E}[X_{i,j}]$.

(g) [3★] Show that the summation from earlier is around $2n \ln n$. **Hint:** Recall that the nth Harmonic number $\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln(n)$.

N7 [3★] If you know calculus, you can also use it to show that the expected number of comparisons in randomized Quicksort is around $2n \ln(n)$. **Hint:** We can place a usually-close upper bound on summations with an integral. Use this to prove the above fact with a Proof by Induction.

N8 Consider an idealized hash function that takes in an object v as an input and outputs $f(v)$, a unique integer from 0 to $M-1$, where M is some integer. A usual assumption made is that the ideal hash function is like rolling a fair die; each of the integers from 0 to $M-1$ could appear with equal probability.

To quickly check if two objects v_1 and v_2 are equal, we check if their hashes $f(v_1)$ and $f(v_2)$ are equal. However, sometimes v_1 and v_2 are not equal, but $f(v_1)$ and $f(v_2)$ are equal anyway; this false positive is called a hash collision.

For each of the following scenarios, the input objects for the hash are strings. Compute for the probability that your program gets a hash collision (and consequently a WA), and find the smallest value of M that will make this probability $< 10^{-6}$. Assume that we can make M arbitrarily large, no need to worry about overflow and whatnot for now.

(a) [2★] Check if b is a substring of a. To do this, you consider all substrings whose length is the same as b, and check if any of them have the hash as b.

(b) [4★] Check if a has any substrings of length at least k that occur more than once in the word. For instance, `banana` has the substring `ana` occur starting from the index 1 and index 3. However in `saging`, all the substrings of length at least 3 are unique.

N9 Suppose a randomized algorithm has one-sided error. It solves a yes-no problem correctly with probability 1 if the answer is **yes**, and probability of at least p if the answer is **no**. Assume that $0 < p < 1$. Suppose the algorithm runs in $\mathcal{O}(f(n))$ time.

In the following, suppose p is fixed and independent of n.

(a) [1★] Show that you can always guarantee the probability to always be ≥ 0.99 by running the algorithm sufficiently many times.

(b) [1★] How many times do you have to run the algorithm to guarantee a correctness probability of ≥ 0.99? of ≥ 0.9999999999?

(c) [1★] Show that you can guarantee the correctness probability to be ≥ 0.9999999999 (or any fixed probability threshold) while still keeping the running time of the algorithm to be $\mathcal{O}(f(n))$. **Hint:** The $\mathcal{O}$ constant depends on the probability.

(d) [1★] Suppose you want to guarantee that the correctness probability is $\geq 1 - 10^{-k}$. Show that you can do so with an $\mathcal{O}(k \cdot f(n))$-time algorithm.

(e) [1★] Using the above results, explain why it's okay to define a randomized algorithm with one-sided error as follows: it's an algorithm that answers correctly with probability 1 if the answer is **yes**, and probability $\geq 2/3$ if the answer is 0. (Or ≥ 0.99 if you like.)

Also, show that adjusting the original algorithm to fit this definition doesn't change its big-$\mathcal{O}$ running time.

This is why computer science books usually define randomized algorithms this way, with fixed threshold (usually 2/3) regardless of problem—they're all equivalent anyway. (This is for algorithms with *one-sided* error—we'll tackle two-sided error in a different exercise.)

In the following, p may depend on n.

(a) [1★] Suppose that $p = c/n$ for some constant c. Show that we can guarantee a probability of $\geq 2/3$ by running the algorithm sufficiently many times.

(b) [1★] What is the running time of the previous algorithm? (This will depend on $\mathcal{O}(f(n))$.)

Hint: You may use the approximation $(1 - 1/n)^n \approx 1/e$ for large n.

(c) [1★] Suppose that $p = c/\log n$ for some constant c. What is the running time now?

(d) [1★] Suppose that $p = c/(n+1)$ for some constant c. What is the running time now?

(e) [1★] Suppose that $p = c/n^3$ for some constant c. What is the running time now?

(f) [1★] Suppose that $p = c/2^n$ for some constant c. What is the running time now?

N10 [2★] Suppose a randomized algorithm has two-sided error. It solves a yes-no problem correctly with probability p_{yes} if the answer is **yes**, and a probability of at least p_{no} if the answer is **no**. Assume that $0 < p_{\text{yes}}, p_{\text{no}} < 1$. Suppose the algorithm runs in $\mathcal{O}(f(n))$ time.

Using the previous exercise, explain how we can simplify the definition by fixing $p_{\text{yes}} = p_{\text{no}} = 2/3$, and the algorithms will have the same big-$\mathcal{O}$ running times.

N11 We'll consider the problem of determining whether an array is sorted without examining all the elements of the array, by allowing some false positives, but not false negatives. Call an array *mostly sorted* if there exists a sorted subsequence with at least 95% of its elements. If the input array is sorted, you must output "sorted". If the input array is not mostly sorted, you must output "unsorted". Otherwise, you can output anything.

(a) [2★] Prove that any deterministic algorithm has runtime $\Omega(n)$. (This is like the opposite of big-$\mathcal{O}$; the problem is asking you to prove that any algorithm has to take at least kn steps, for some fixed $k > 0$.)

(b) [3★] Consider the following binary search algorithm; the input array is `A`.

```python
def binary_search(x, left, right):
    if right - left == 1:
        return left
    median = (left + right)//2 # floor division
    if x < A[median]:
        return binary_search(x, left, median)
    else:
        return binary_search(x, median, right)
```

Let x_1 and x_2 be distinct elements in the array. Show that binary search on x_1 gives a smaller result than binary search on x_2 iff $x_1 < x_2$, even if the array isn't sorted.

(c) [8★] Provide an $\mathcal{O}(\log n)$ randomized algorithm that is correct with probability at least 75%.

Hint: What can you say about the number of distinct indices that can be the result of binary search?

3.2 Coding problems

The problems about floating-point use draw from a variety of different topics, with their binding factor being that you have to deal with floating point numbers of some kind in their implementation. The items intended to be about randomization may also have deterministic solutions, which are equally valid; randomization is only one possible approach.

C1 Find two input triples (P, A, B) with floating-point coordinates such that (1) and (2) are both nonzero and have different signs, even if we compute the sign with the following code (with $\varepsilon = 10^{-8}$):

```cpp
const double EPS = 1e-8;
int sign(double v) {
    if (v >= +EPS) return +1;
    if (v <= -EPS) return -1;
    return 0;
}
```

You may use a computer.

Hint: Try generating lots of random cases! You will want to find points P, A and B that are very nearly collinear.

S1 [5★] **Google Code Jam 2018 Qualifying Round – Go, Gopher!:** https://codingcompetitions.withgoogle.com/codejam/round/00000000000000cb/0000000000007a30

S2 [5★] **Google Code Jam 2019 Round 3 – Zillionim:** https://codingcompetitions.withgoogle.com/codejam/round/0000000000051707/0000000000158f1a

S3 [5★] **Google Code Jam 2019 Round 2 – Pottery Lottery:** https://codingcompetitions.withgoogle.com/codejam/round/0000000000051679/00000000001461c8

S4 [8★] **Pizza Problem:** https://open.kattis.com/problems/pizzaproblems

S5 [8★] **Google Code Jam 2018 Round 3 – Name-Preserving Network:** https://codingcompetitions.withgoogle.com/codejam/round/0000000000007707/000000000004ba29

S6 [8★] **Code Jam to I/O for Women 2019 – War of the Words:** https://codingcompetitions.withgoogle.com/codejamio/round/0000000000050fc5/0000000000054ea5

S1 [3★] **Annoying Present:** https://codeforces.com/problemset/problem/1009/C

S2 [4★] **Nature Reserve:** https://codeforces.com/problemset/problem/1059/D

S3 [6★] **Seating of Students:** https://codeforces.com/problemset/problem/906/B

S4 [9★] **DZY Loves FFT:** https://codeforces.com/problemset/problem/444/E

S5 [10★] **Convex Quadrilateral:** ICPC Live Archive 7853

S6 [10★] **Smart Thief:** https://codeforces.com/gym/102001/problem/C

S7 [15★] **Gena and Second Distance:** https://codeforces.com/problemset/problem/442/E