class 2: Random number generation

I would like to know how long does it take to find a number in an array, on average.

Not a very exciting problem, I admit. In fact, since the numbers in the array are not sorted, there does not exist a smart algorithm to solve the problem. The algorithm is just this: iterate over every element in the array, and compare each element in the array with the searched for value. However, even despite the fact that the problem is not exciting, the way in which we shall go about finding the average is a roller coaster of problems.

The Input

First problem, which I had carefully and purposefully omitted from the previous class, is this: what is the input data to our problem?

If You read the previous class again, You will only see that I have specified that we have some operations to perform on some numbers. I have also never explicitly stated what operations are possible: I merely gave an example with four basic two-argument operands, `+`, `-`, `*` and `/`. If there were more operands, and the operands were far more complicated (why not the power operation, or the Ackermann function?), then computation would take far longer. In fact, if the only legal operands were to be `+` or `-`, I bet that the code might be faster.

Apart from that, I have never explicitly stated what numbers are we operating on. In fact, all the numbers in my data files were single digit integers, with the omission of 0. Not only that, if You carefully read my code, You will notice I have used an integer to store the result. If I had used floating point numbers, then the task would take longer to compute, since floating point operations are usually more difficult than integer operations -- and that also depends on the precision of the numbers (doubles or floats? int vs int64_t?).

In any case, let us specify this for now: we are searching for a pseudo-random integer in an array of pseudo-random integers.

Generating pseudo-random numbers

Let us start by writing a program which prints 10 integers. But the integers have to be different each time we run the program.

We begin by doing this in the *wrong* way. You may have been taught this in some other class, or You may have found it on the Internet. Please do not use this. Code that uses this for the project will automatically need to be redone.

rand/srand

Here is Your first non-deterministic program (maybe, that depends on what was taught on other courses).

```
#include <iostream>
#include <ctime> // or time.h
#include <cstdlib> // or stdlib.h

int main() {
    srand(time(NULL));

    for (int i = 0 ; i < 10 ; i++) {
        std::cout << rand() << std::endl;
    }
}
```

This program outputs values which, by design, differ in each run of the program.

The code is quite simple. We call srand() with time(NULL) as argument, and then whenever we want a random value we call rand(). Nice and simple.

Let us run it two times, and let us see the results:

```
 > ./rand1
934097396
878244275
2038151876
1423346524
344687531
1354406865
1353510850
1791202072
623524354
901131502
 > ./rand1
623262126
685626103
1604335085
2102455929
1269811923
1816893865
1044764848
269120568
192127562
1070270340
```

The results are random, so what is the problem?

First, the random number range is implementation defined. So, the numbers that You see generated might differ strongly from what I have presented. The standard defines that `rand()` returns a pseudo-random number between `0` and `RAND_MAX`, and it defines

`RAND_MAX` to be at least 32767. For me, `RAND_MAX` is 2147483647. You can print it out easily and check on Your system.

```
std::cout << RAND_MAX << std::endl;
```

If Your `RAND_MAX` returns 32767 (as it returns for MinGW bundled with older versions of Code::Blocks installed on university machines, or DevC++ inexplicably still being used by some people), then that already rules out `rand()` as a good source of pseudo-random numbers for our purposes.

seeding

`srand` is used to provide the pseudo-random number generator with initial state. This is called *seeding*. It is typically done only once. Each subsequent call to `rand()` modifies the state, and returns a value based on that state. `rand` is deterministic: if the state is known in advance, then not only the next random number is known, but also all future numbers can be perfectly predicted. So if we were to seed the state with the same value, the program ceases to generate different values each time it is run: it will always produce the same values.

This brings us to the second problem: we initialise with `time(NULL)`. `time` returns seconds since the Epoch (approximately). Which means, that if we run the same program twice within one second (which is not difficult to do), we will generate the same results, twice!

```
 > ./rand1
1116966242
1214783642
717080516
522655353
1186867430
153721984
395963693
450368835
577254995
2058453718
 > ./rand1
1116966242
1214783642
717080516
522655353
1186867430
153721984
395963693
450368835
577254995
2058453718
```

You may not believe that this is a problem. But every year I get the following code:

```
int main() {
    for (int size = 10 ; size < 100000 ; size *= 10) {
        for (int times = 0 ; times < 1000 ; times++) {
          int array = new int[size];
          srand(time(NULL));

          // fill the array with some random numbers generated with rand()

          // measure the time of some algorithm

          delete [] array;
        }
    }
}
```

If the inner loop takes less than a second to compute, then the inner loop will generate the same random numbers multiple times. This automatically disqualifies the code, because instead of keep testing the same data. In fact, for very small sizes, the whole batch of 1000 computations with the same size (for instance, for size 10) will operate on the same numbers!

state and generation

The state that `rand` uses to generate pseudo-random numbers has finite size. Therefore, since `rand` is also deterministic, it will eventually end up in a cycle.

This is not a bad thing -- in fact, this is expected. But if You generate enough numbers, the sequence that was generated repeats itself perfectly, with a certain period. The POSIX standard defines that the period of the cycle needs to be at least 2^{32}, which is large enough for simple purposes.

But let us go back to the previous class for a second. In the largest example, I computed 100 million operations, and repeated the calculation 100 times. If the cycle is precisely 2^{32}, and I generated them pseudo-randomly using rand(), then the cycle would have inevitably repeated itself, since $10^8 \cdot 10^2 > 2^{32}$. If this happens, then Your results are inevitably skewed: You repeat the same operations on the same data, instead of performing them on different pseudo-random values. This happens faster than You might think: on Code::Blocks installed on the university machines, You repeat the same numbers after 22 seconds of generating pseudo-random numbers using rand().

The reason for this behaviour is the pseudo-random number generator used in the C library bundled with MinGW used in conjunction with Code::Blocks. It uses a linear congruentional generator -- the generator gives us a pseudo-random permutation of all numbers in range from 0 to 2^{31}.

You can find an example of such a generator in `man 3 rand`.

```
EXAMPLE
       POSIX.1-2001 gives the following example of an implementation of rand()
and  srand(),  possibly
       useful when one needs the same sequence on two different machines.

           static unsigned long next = 1;

           /* RAND_MAX assumed to be 32767 */
           int myrand(void) {
               next = next * 1103515245 + 12345;
               return((unsigned)(next/65536) % 32768);
           }

           void mysrand(unsigned int seed) {
               next = seed;
           }
```

If You use a more modern C standard library, for instance on any recent Linux, or on Visual Studio, You *should* be fine. In fact, modern rand() uses far more complicated generators (such as the additive feedback generator).

the distribution and the range

What is the probability of generating particular values using the pseudo-random number generators?

For instance, if we flip an honest coin, the probability of getting tails should be the same as getting heads. If we throw one honest die, the probability of getting each value on the die is the same -- the probability distribution of values is uniform. But if we throw two honest dice, the probability of the sum of values on the dies is not uniform: getting a 7 is more likely than getting a 2.

The distribution of values returned by `rand()` is uniform: You are as likely to get 1 as You are to get 2, 3 or any other value from 0 to `RAND_MAX`. But suppose You wish to generate values from 0 to 99, or -49 to 50. The usual (and very subtly *wrong*!) way to do it is this:

```
    int zero_to_99 = rand() % 100;
    int neg49_to_50 = (rand() % 100) - 49;
```

But this changes the distribution and slightly skews Your result!

Assuming that `RAND_MAX` is 32767, and we wish to get random numbers from 0 to 99. If we divide modulo 100, thanks to the pigeonhole principle, the probability of getting 1 is now slightly higher than getting 99. Even despite the fact that the original distribution was uniform, once we scale it down to [0, 99], the distribution ceases to be uniform! The difference is not large, especially if `RAND_MAX` is 2147483647, but such trivial changes to the distribution may show an effect where there is none -- but that is usually for very

professional situations. Since You are unlikely to ever be present in such a situation, then `(rand() % range_length) - range_start` *should* be enough if `RAND_MAX` is large.

If You are ever stuck with using a uniform number generator such as `rand()`, and need to scale it such that the distribution is uniform, then do the following:

```
    int range_length = 100;
    // assuming that range_length < RAND_MAX
    int good = (RAND_MAX / range_length) * range_length;

    int val = good + 1;
    while (val > good) {
        val = rand();
    }

    val %= range_length;
```

But what if You want a different distribution? If You are still stuck using a uniform number generator, You may try rejection sampling or an inverse transform sampling if the cumulative distribution function is known.

The <random>

We have discussed various issues long enough. But now let us get back to C++.

C++ provides mechanisms to alleviate some of the above concerns. The `<random>` library provides a high level interface to random number generation. But it also may create some headaches of its own, which we will discuss shortly.

First, the good things.

The `<random>` library provides the user with access to various random and pseudo-random number generators. It also allows the user to generate numbers according to various well-known probability distributions. It is also quite easy to make a generator per thread, so that values generated by each thread are isolated from other threads without the need for synchronisation. It provides mechanisms for various types of seeding, it has large state sizes and very large periods, all of which is generally configurable. But You generally should not tamper with the configuration, unless You know what You are doing.

The basic usage is this:

```cpp
#include <iostream>
#include <random>

int main() {
    std::random_device rd;
    std::mt19937 gen(rd());

    std::uniform_int_distribution<int> dist(-100, 100);

    for (int i = 0 ; i < 10 ; i++) {
        std::cout << dist(gen) << std::endl;;
    }
}
```

Let us slowly digest what is being done, step by step:

First, `std::random_device rd` is a declaration of our source of a seed. A call to `rd` provides a uniform random number, and we use it to seed to an instance of `std::mt19937`. `std::mt19937` is a pseudo-random generator -- it generates pseudo-random numbers, like `rand()`. However in contrast to `rand()`, it is an object, and so it is clear where the state of the generator is located, and can be managed (`rand()` stores state in global variables). We may call it directly, `gen()`, which returns a 32-bit pseudo-random integer (this also changes the internal state of the object). Just like `rand()`, it is deterministic -- if You run the code twice with the same seed, You will get exactly the same results.

And finally, the `std::uniform_int_distribution dist(-100, 100)`. `dist` scales a pseudo-random integer to fit the uniform distribution. We give it two values: `min` and `max`. In short, by calling `dist` on `gen`, we get a random integer from the range `[min, max]` with uniform probability, using `gen` to generate the integer.

There are various different distributions. You can use `std::uniform_real_distribution` when You need a floating point number. There are objects for normal and lognormal distributions, Poisson, gamma and many more.

There are also various different generators that You could use. You could use `std::mt19937_64` to get 64 bit integers instead of 32 bit integers. For instance, we could also use the `std::minstd_rand` if we wanted a linear congruential generator, or others. `std::mt19937` is a safe bet -- if in doubt, use `std::mt19937`.

the headaches

Generally there are very few headaches with the above code. It is generally safe to use. The only drawback is the size of the generator (it uses several KB of stack space -- seems small, but depending on usage might be significant).

However, there is one thing that You might ruin Your day at some stage in life. And that thing is `std::random_device`.

Don't get me wrong, `std::random_device` is in most cases extremely nice, safe to use and generally very good.

The main problem with `std::random_device`, is that it might not be random -- at all. In fact, each call to `std::random_device` might return the same, deterministic value (for instance 1). In other situations, the value returned by `std::random_device` might be only pseudo-random (which might be good enough if the pseudo-random number generator was seeded properly, for instance by the operating system, or might be very bad if it was not).

In some other cases, `std::random_device` might call on the operating systems method of generating genuine random integers. `std::random_device` might even call on hardware random number generators, which is fantastic! For instance, `RDRAND` or `RDSEED` instructions might be used to get a random number from the Intel hardware random number generator.

Hardware random number generators use entropy from physical phenomena to generate truly random numbers (Intels DRNG uses thermal noise in silicon -- but other methods are also available). Which might in very specific circumstances be a problem -- I have been in such a situation, so I shall explain, for posterity.

A hardware random number generator produces high quality random numbers -- and to do so it needs to gather some entropy from physical phenomena. This makes it slow -- it is still imperceptibly fast, but it is slower than generating pseudo-random numbers. But there is a problem: if You have a lot of threads and they all simultaneously call for a large amount of random numbers, then the hardware random number generator does not produce results fast enough. And since in libstdc++ the `__x86_rdseed` function is implemented as a spinlock (try 100 times, and if it fails, throw an exception), then You get a situation that is very hard to diagnose (especially given that the exception that is crypic at best: `terminate called after throwing an instance of 'std::runtime_error' what():  random_device: rdseed failed`). But this is a very isolated, extremely specific example -- but it might happen. It might not even happen on Windows: on Windows calls to `std::random_device` might block the thread until the hardware generates enough entropy to provide a random number.

So in short: `std::random_device` might not be random at all, might not be genuinely random or might in very specific situations throw a very surprising exception.

More on this here.

code::blocks and dev-c++

I have been advising against using the Code::blocks installed on the university machines, and I shall do so again: the MinGW installed has one more problem, this time with `std::random_device`. In old versions of MinGW, bundled with Code::blocks and dev-c++, the random_device always returned the same, deterministic value for each call. So all runs of code were completely deterministic, which kind of defeated the purpose of writing a non-deterministic program.

If You are using code::blocks, check if `std::random_device` returns the same, deterministic value. If every run of the program returns the same numbers in the same order, despite using `std::random_device` to seed the generator, use `std::chrono::high_resolution_clock::now().time_since_epoch().count()` to seed the `std::mt199937` instead. This is equivalent to seeding with `time(NULL)`, but hopefully with higher precision.

so, anyway...

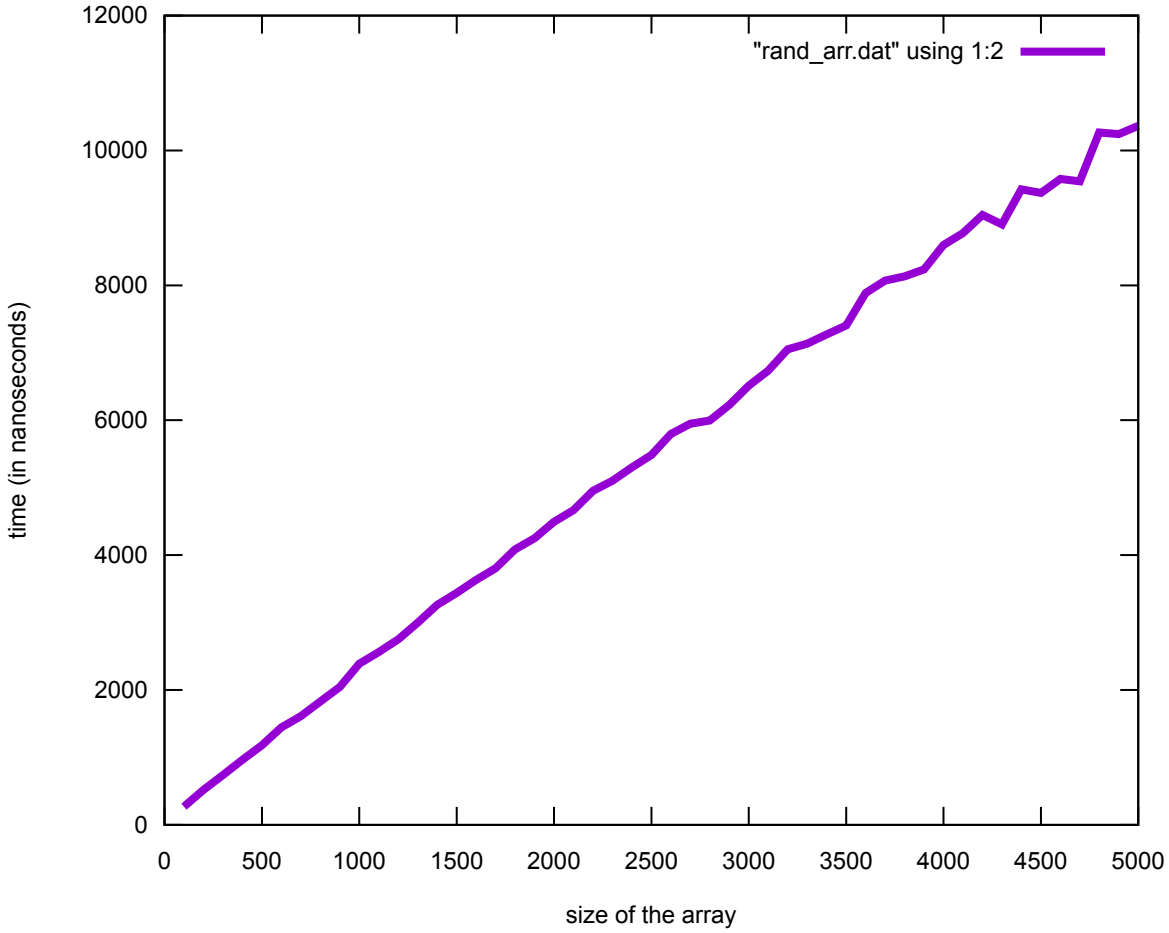
Even despite all the mean things that I said against `<random>`, I still advise using `std::random_device`, `std::mt19937` and distributions from `<random>` to generate pseudo-random numbers. In fact, I shall force You to do so, unless You can give me a very good reason not to us it (choosing to program only in pure C is a good reason).

Back to our original problem: I want to search for a pseudo-random number in arrays of pseudo-random numbers.

Now that we know how to properly generate random numbers, let us try this way:

```cpp
#include <iostream>
#include <chrono>
#include <random>

int main() {
    using std::chrono::nanoseconds;
    std::random_device rd;
    std::mt19937 gen(rd());

    std::uniform_int_distribution<int> dist(-10000, 10000);

    int max_times = 1000;

    for (unsigned size = 100 ; size < 5000 ; size += 100) {
        nanoseconds total(0);
        int *haystack = new int[size];

        for (int times = 0 ; times < max_times ; times++) {

            for (unsigned j = 0 ; j < size ; j++) {
                haystack[j] = dist(gen);
            }

            int needle = dist(gen);

            auto start = std::chrono::steady_clock::now();

            for (unsigned j = 0 ; j < size ; j++) {
                if (needle == haystack[j]) {
                    break;
                }
            }

            auto end = std::chrono::steady_clock::now();

            nanoseconds time_taken(end - start);

            total += time_taken;

        }

        total /= max_times;

        std::cout << size << " " << total.count() << std::endl;

        delete [] haystack;
    }
}
```

I get the following results, organised neatly in a graph:

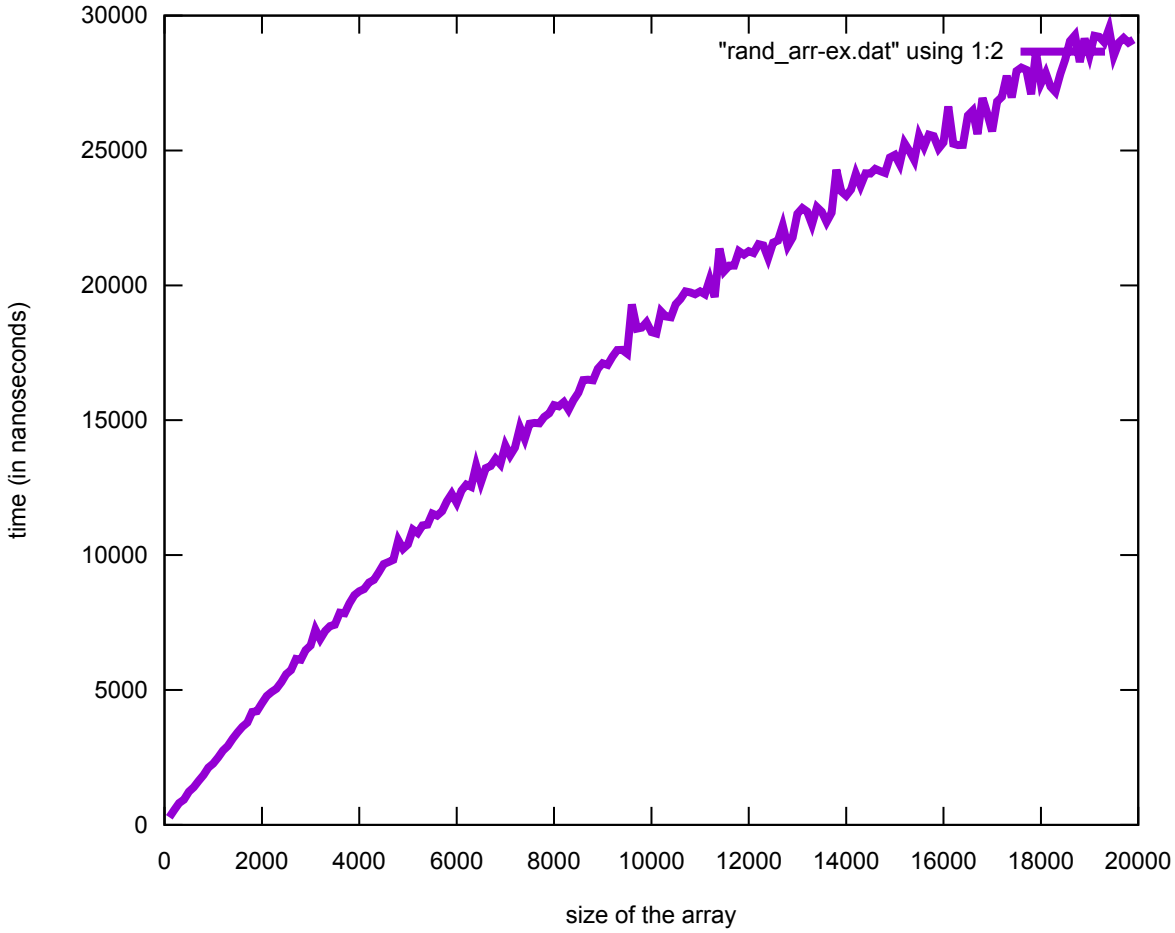


Now, this looks "bumpy", and it is rightly so: this time, searching for a number may take any time (it might be in the first position in the array, or it might not be in the array at all -- it's all random). But we average over many runs, the outliers cancel each other out, and so it looks generally linear. In fact, it should look linear. What we do is merely iterate over an array, so the time it takes for the algorithm to finish should in some way depend on the size of the array. Since we do only one pass of the loop, if the array has n elements, it should take at most $c \cdot n$ to find a number in the array, or check all values in the array and conclude that it is not there.

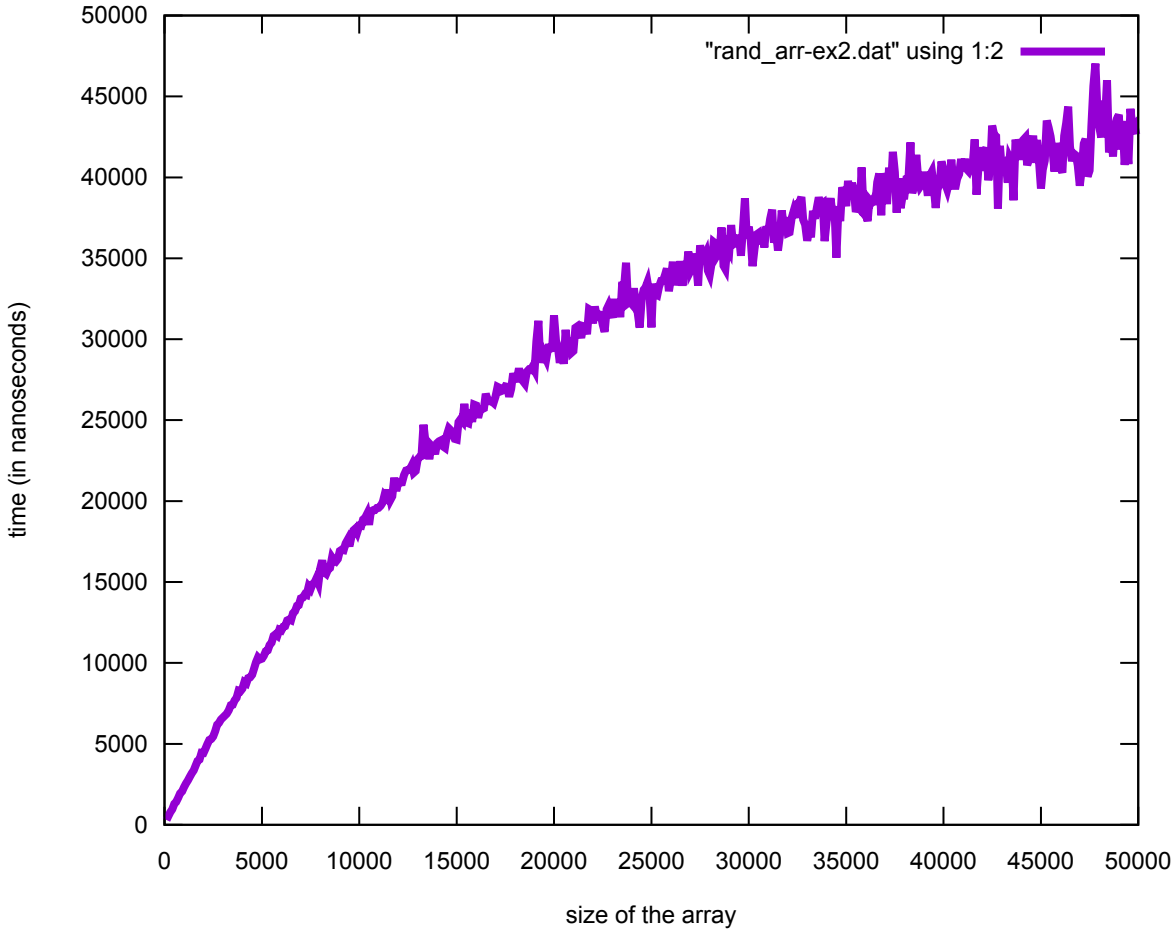
On average it takes 205.28 nanoseconds longer with each run to search for a number in an array for each 100 elements in the array. So we can predict that for one million elements it should take about 0.2s to get a result.

But it is not so. In fact, it takes about 45982 nanoseconds, which is approximately 0.000046, or about 4350 times faster than expected.

Let us see what happens when we extend the size of the array:



And once more:

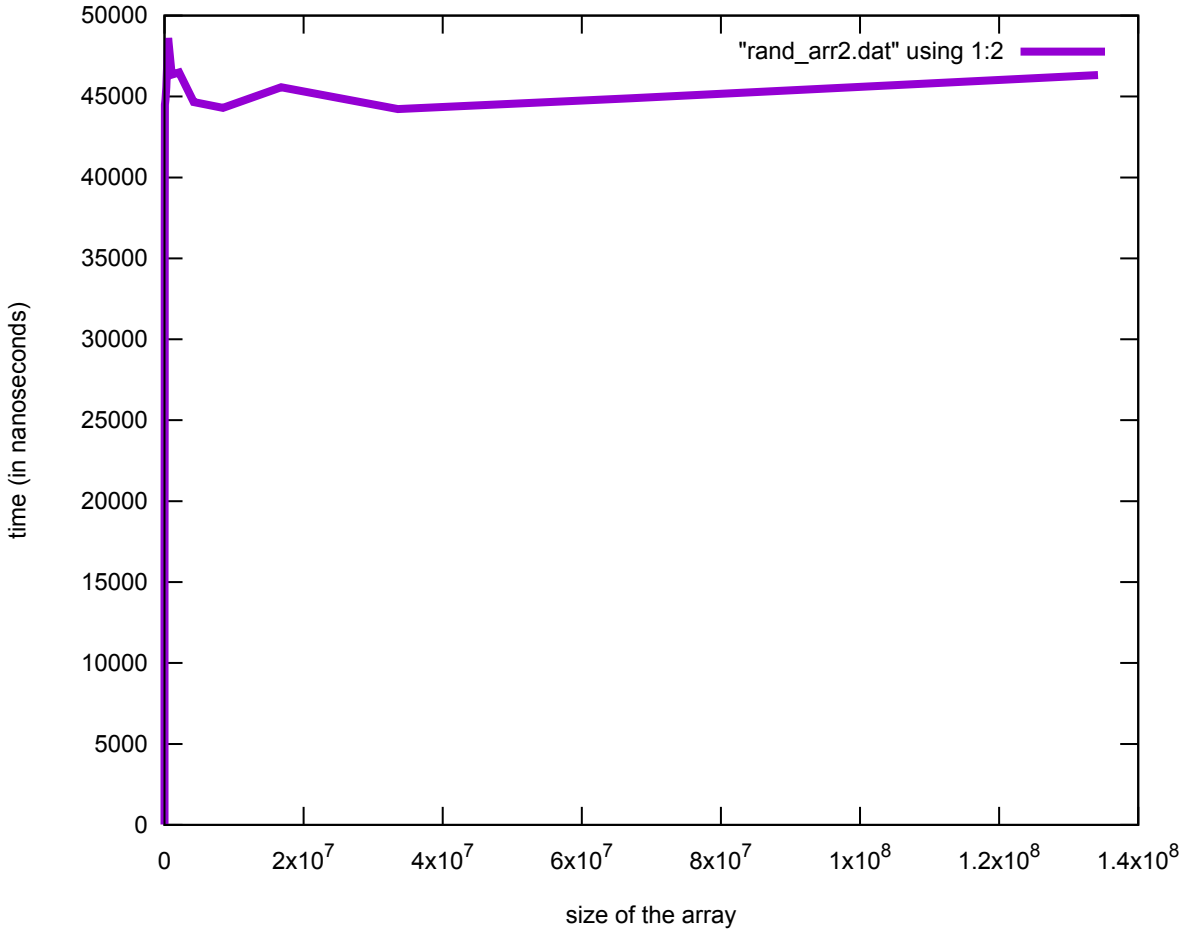


Clearly our previous assumption on the linearity was wrong!

An let us see what happens for larger sizes:

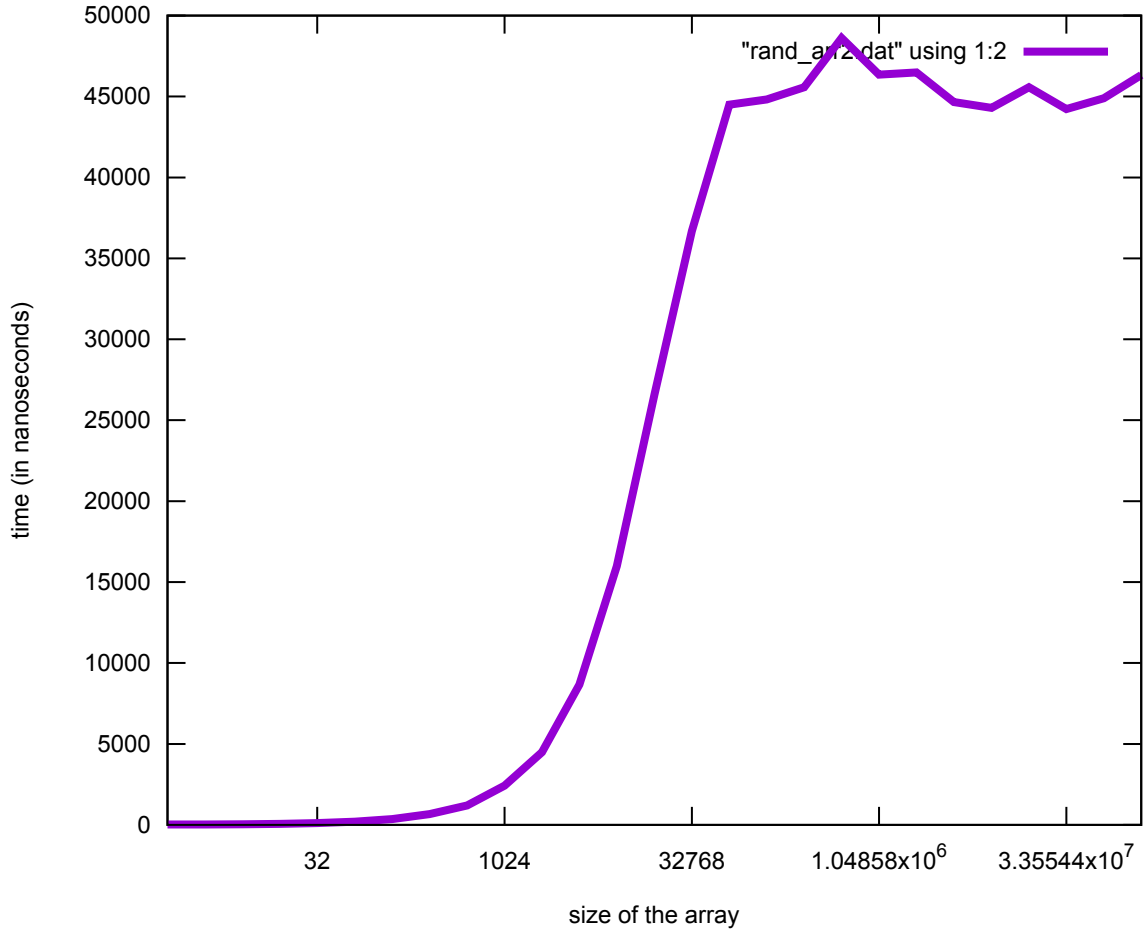
```
size     time (ns)
======== =========
2               23
4               27
8               35
16              57
32             115
64             195
128            353
256            664
512           1203
1024          2422
2048          4493
4096          8673
8192         15993
16384        26594
32768        36687
65536        44498
131072       44822
262144       45573
524288       48614
1048576      46356
2097152      46486
4194304      44656
8388608      44299
16777216     45576
33554432     44220
67108864     44898
134217728    46327
```

And let us also visualise it:



The above is quite unreadable, although it shows that there exists a certain cut off point, where the time necessary to find the number stops changing. This is also visible quite clearly from the raw numbers presented above the graph.

Let us change the x-axis into logarithmic scale to visualise what happens for smaller numbers as well:



In the first part, the time grows linearly (it looks exponential, but that is because the x-axis is logarithmic). Then it reaches a certain spot (around somewhere between 32768 and 65536), where the time stops growing at all.

back to the problem

The reason for this is simple: unfortunately the problem was described in an imprecise manner, or rather: I purposefully left a part ambiguous to illustrate some dangers.

Danger 1: in the first graph, we saw that the time behaved linearly, and we assumed that it would remain so. This happens often. We deduced conclusions based on a small sample. We almost always deduce on incomplete data, because very rarely the data can be complete. But this time we only checked what happens for small arrays (up to 5000). Sometimes it is enough -- for testing operations it would be enough. But often it is not.

Danger 2: increasing the size of the array changes the problem -- so the results collected concern different problems. The strange behaviour here arose because of the chosen distribution for the random numbers. Consider what would happen if I only generated random numbers from a range [0, 10]? It should be quite obvious now: in an array of 1000 elements randomly generated from a range [0, 10], it is statistically extremely improbable that some number from range [0, 10] is not present. If so, then we are almost guaranteed to find a number, if we search for a number from a range [0, 10]. After some size of the array,

it will not matter what the size of the array is: we are certain to find any number from the range within first values in the array. Therefore the time necessary to find it will not grow with the size of the array, after a certain value.

Clearly the Danger 2 is a result of choosing the distribution with range [-10000, 10000]. But this problem is not merely solved by increasing the distribution to, say [-1000000, 10000000]. If we do this, then it is unfair on the small arrays. Consider we generate an array of 10 random integers from the range [-1000000, 10000000], and we search for an integer from the same range. The chances that it will be found are very small -- You can try it out 1000 times, and You still might not find it at least once.

We need to think more about the data. We should specify better what the data should look like (both in the problem and in the code that tests it). Without a good description of how the input data looks like, different people might reach different conclusions, since they will be free to choose the distribution of their input data.

I propose that we specify that the range from which we generate an array to be tied to the size of the array. If the array is of size n, we generate integers from the range [-n, n]. In such a situation, we should have approximately 61% chance not to find the searched for number.

TODO

Please try to fix the code to correctly generate integers from the range [-n, n].

Generate times for arrays of various sizes. Organize them in a graph, notice its shape.

Try to predict how long it will take to find a random number from range [-n, n] in an array of n random values from range [-n, n]. Verify that claim -- generate times for the predicted size, and maybe arrays of larger values.

Change the ranges: try a wider range (say, [-10n, 10n], and a much smaller range [0, n/2]). How did the results change? Try to plot the results for several ranges on one graph.