10. Performance measurement

```
./memleak 1024 1500000
```

Following memory usage with top: At about 1 000 000 steps, 50% memory usage is exceeded.

Runs with constant value of *m*, varying value of *N*:

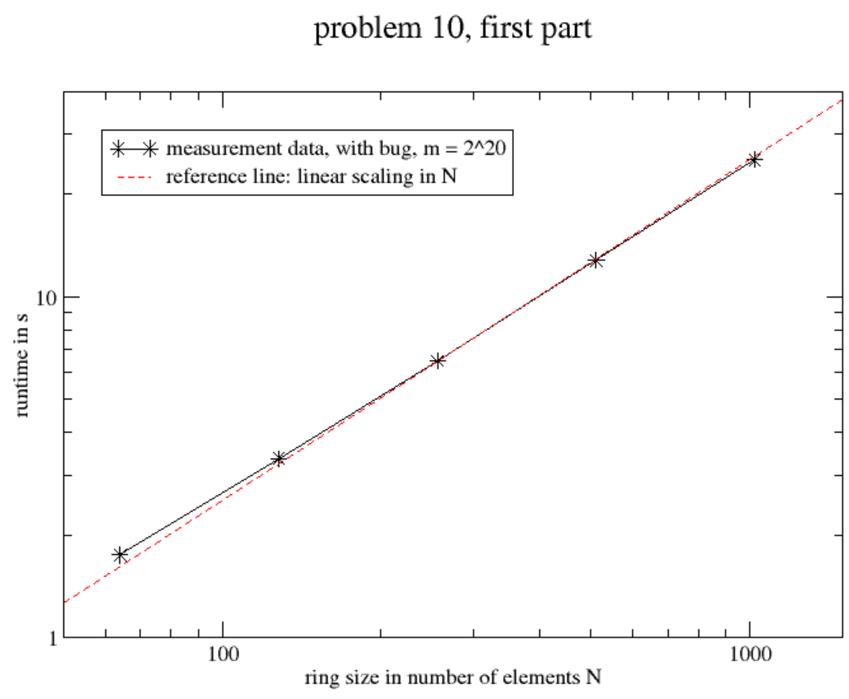
Setting the number of steps to $m = 2^{20} = 1\,048\,576$.

Scaling down in steps of factor 4, which is enough to see the trend.

```
                      N      m
time ./memleak 1024 1048576      25.22 s
time ./memleak 512 1048576       12.78 s
time ./memleak 256 1048576       6.482 s
time ./memleak 128 1048576       3.355 s
time ./memleak 64 1048576        1.747 s
```

Taking note of the time measurement (using the "real" output value).

Result: We can see that at constant *N*, the run time is approximately proportional to *m*.

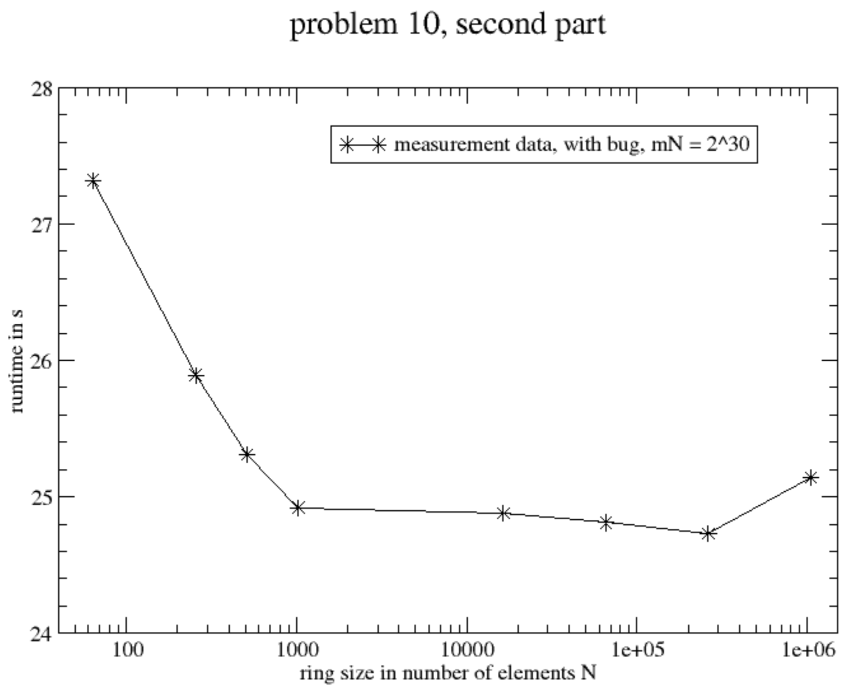


Runs with constant value of *mN*, varying value of *N*:

Setting product of number of steps and chain length (ring size) to $mN = 2^{30}$.

```
                    N      m
time ./memleak 64 16777216         27.32 s
time ./memleak 256 4194304         25.89 s
time ./memleak 1024 1048576        25.31 s
time ./memleak 4096 262144         24.92 s
time ./memleak 16384 65536         24.88 s
time ./memleak 65536 16384         24.81 s
time ./memleak 262144 4096         24.73 s
time ./memleak 1048576 1024        25.14 s
```

Result: With constant *mN*, there is a weak decreasing trend as a function of *N*.



11. Fixing the memory leak

Matching new and delete in same portion of code

Let us rewrite crw::step such that it only uses memory on the heap that it deallocates itself.
This makes sense, in this way the new and the matching delete are in the same piece of code.

```
void crw::step(long size, float config[])
{
  assert(size >= 2);
  // temporarily copy the pre-existing configuration
  float* previous = new float[size]();
  for(long i = 0; i < size; i++) previous[i] = config[i];
  // first, let the chain contract: each element is attracted by its neighbours
  for(long i = 1; i < size-1; i++)
    config[i] = 0.5*previous[i] + 0.25*(previous[i-1] + previous[i+1]);

  // special cases, first and last element
  config[0] = 0.5*previous[0] + 0.25*(previous[size-1] + previous[1]);
  config[size-1] = 0.5*previous[size-1] + 0.25*(previous[size-2] + previous[0]);

  stochastic_unit_step(size, config);  // actual random walk step
  shift_centre_to_origin(size, config);  // shift such that the average remains zero
  delete[] previous;
}
```

This version of crw::step only allocates temporary memory for storing the old configuration.
The array containing the configuration is overwritten, so that at the end, it contains the new one.

Changing main() so that it correctly calls the updated crw::step function

In main, we will need two arrays: One for the present and one for the extreme configuration.

Before:

```
  float* present_configuration = new float[size];
  float* extreme_configuration = present_configuration;
```

After:

```
  float* present_configuration = new float[size]();
  float* extreme_configuration = new float[size]();
```

Now, we can simplify the function call in the loop where all the computation is done. This is just to match the specification of crw::step. Note that we changed its return value from float* to void.

Before:

```
    present_configuration = crw::step(size, present_configuration);
```

After:

```
    crw::step(size, present_configuration);
```

The array extreme_configuration keep tracks of the configuration with the greatest elongation so far. We need to copy the present into the extreme configuration to update it to a new record.

Before:

```
    if(present_elongation > extreme_elongation) {
        extreme_configuration = present_configuration;
        extreme_elongation = present_elongation;
    }
```

After:

```
    if(present_elongation > extreme_elongation) {
        for(long i = 0; i < size; i++) extreme_configuration[i] = present_configuration[i];
        extreme_elongation = present_elongation;
    }
```

At the very end of main(), we should matching deletes to the two new's higher up.
It does not make a real difference, since it is at the end of the code, but is recommended style.

```
  delete[] present_configuration;
  delete[] extreme_configuration;
```

Now, for every "new" in the program, there is a matching "delete" in the same piece of code.

Runtime comparison

First, using *top* we confirm that indeed the memory leak has been fixed.
Then, let us run a few instances with constant $mN = 2^{30}$.

	N	*m*	memleak	fixed	**speedup**
time ./memleak-bugfix	64	16777216	27.32 s	27.99 s	0.976
time ./memleak-bugfix	1024	1048576	25.31 s	26.05 s	0.972
time ./memleak-bugfix	16384	65536	24.88 s	25.59 s	0.972
time ./memleak-bugfix	262144	4096	24.73 s	25.83 s	0.957

Result: The fixed version is actually slower, because it needs to copy more data items.

12. Optimizing the code

Optimization by hand

First, we don't change the compiler settings, only the code, to see what improvements we can obtain from more efficient coding. We keep track of the runtime for $N = 2^{10}$, $m = 2^{20}$. On my work laptop, this took **27.32 s** for the original code which had a memory leak. The fixed code did not have a memory leak any more, but performance had deteriorated to **27.99 seconds** runtime.

Copying is now done element by element, even though the content to be copied is contiguous in memory, since it is in an array. In such a case we can copy more efficiently using std::memcpy, from the <cstring> part of the C standard library.

Before:

```
        for(long i = 0; i < size; i++) extreme_configuration[i] = present_configuration[i];  // in main
```

and

```
  for(long i = 0; i < size; i++) previous[i] = config[i];  // in crw::step
```

After:

```
       std::memcpy(extreme_configuration, present_configuration, size*sizeof(float));  // in main
```

and

```
  std::memcpy(previous, config, size*sizeof(float));  // in crw::step
```

We are now down to **24.44 seconds** runtime.

Another simple improvement can be made in the function shift_centre_to_origin.

Before:

```
    for(long i = 0; i < size; i++) config[i] -= sum/size;
```

After:

```
    float shift = sum/size;
    for(long i = 0; i < size; i++) config[i] -= shift;
```

It does not make sense to compute sum/size over and over in the same loop, when we can simply compute it once. With this fix, we are now at **24.06 s** runtime, or a **speedup 1.135** with respect to the original code. That is not much, and certainly not the best that can be reached.

There would also be a major potential for improvement in using the random number generator. But I am not touching this for now as it looks rather complicated.

Compiling the code for production

We now turn on the **compiler flag -O3** for a thoroughly optimized code which should be faster.

```
g++ -O3 -c -o chain-random-walk.o chain-random-walk.cpp
g++ -O3 -c -o chain-sampling.o chain-sampling.cpp
g++ -O3 -o chain-production chain-random-walk.o chain-sampling.o
```

(Remark: The file that used to be called memleak.cpp was renamed to chain-sampling.cpp.)

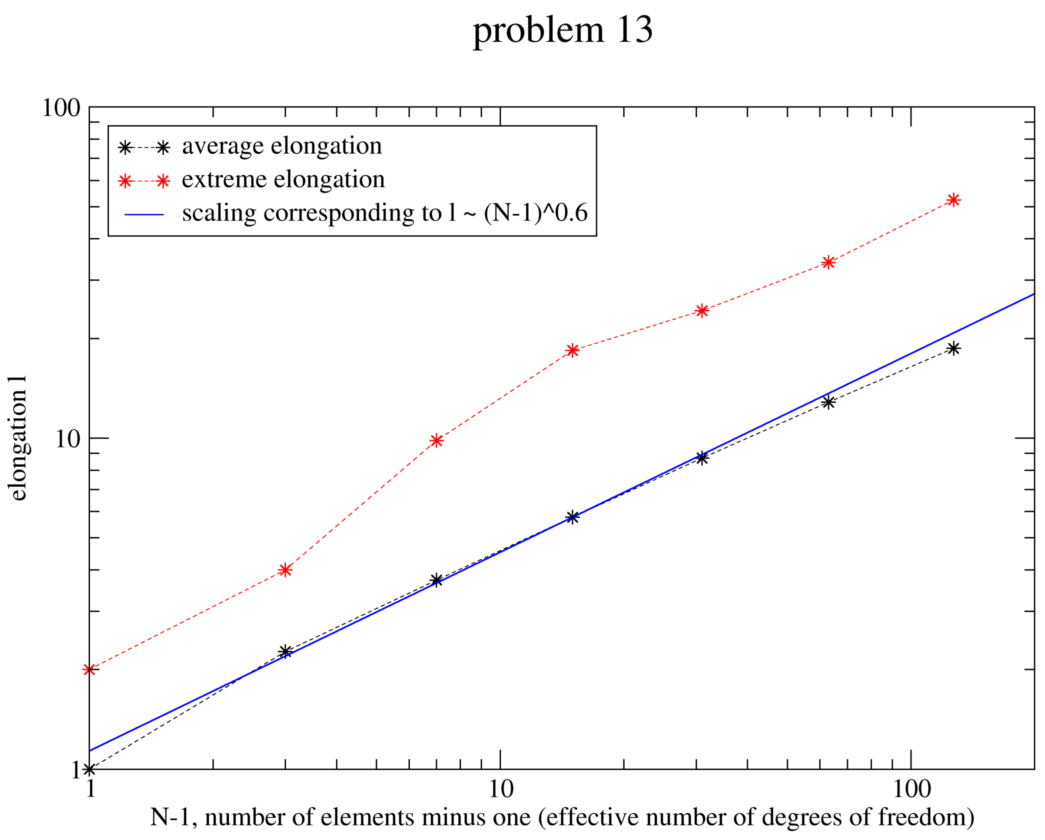
And indeed it is somewhat faster:

	N	*m*	memleak	production	**speedup**
time ./chain-production	64	16777216	27.32 s	17.88 s	1.528
time ./chain-production	1024	1048576	25.31 s	17.70 s	1.430
time ./chain-production	16384	65536	24.88 s	17.56 s	1.419
time ./chain-production	262144	4096	24.73 s	17.68 s	1.399

So where from optimization by hand we obtained 1.135 speedup, with -O3 this became 1.430.

13. Behaviour of average and extreme elongations for the ring with *N* elements

It could be more interesting to do a different series of runs for this problem, with smaller *N*. Runs for varying element number *N*, with a constant number of steps $m = 2^{24}$, yield:



The problem of the average elongation of a ring-like chain is similar to that of the radius of gyration R_g for polymer molecules, where **Flory theory** yields a scaling $R_g \sim N^{3/5}$. In view of this, the present results are unsurprising. They suggest: The simulated system behaves like a polymer.

See for example https://en.wikipedia.org/wiki/Polymer_physics#Solvent_and_temperature_effect

14. Error bars / uncertainty of the results

There are many good (and some poor, but popular) ways to do this. It will be interesting to hear what different people are used to as their favourite approach to estimating an error.

To give a concrete recommendation, the slides from my DAT121 lecture on Flyvbjerg-Petersen block averaging summarize one of these methods (the relevant parts are slides no. 20 to 27).