Confirming Big Oh, Reading

It's one thing to come up with a nice algebraic or differential calculus derivation, or a well reasoned conclusion to arrive at the big oh for an algorithm or program, but it's quite another thing to have to *prove* it. When it comes right down to it, what really matters is how fast it runs.

The approach to confirming big oh is to write a test program and run it with different n values, and the see if the timing results match your expectations. For example, if you wish to check whether your Stack::push function is O(1), you'd expect the same runtime for any n. But for a nested for-loop sort, you'd expect to see the runtime *quadruple* every time n *doubles*.

Format Of The Results

The timing test that we will apply in this class will be based on 4 different runs -- not separate runs of the test program, but in 4 different cycles of a loop in one run of the program. The first cycle will start with n equal to some large number. The next cycle doubles it, then the next doubles it again, and the last doubles it yet again. The ending n will be 8x the starting n. Since the largest (signed) int is 2 billion in C++, the starting n cannot be larger than one eighth of that value, or 250 million. It's usually a lot less, though, to keep the overall runtime of the test reasonable.

The *expected* big oh also needs to be specified. Here are sample results for an expected $O(n^2)$ starting with n=1,000 in the first cycle, showing the actual and expected number of seconds for each cycle:

n	n^2	actual	expected
1000	10^6	0.1324	0.1324 (actual)
2000	4×10^6	0.5436	0.5296 (0.1324 x 4)
4000	1.6×10^7	2.0784	2.1184 (0.1324 x 16)
8000	6.4×10^7	8.6120	8.4736 (0.1324 x 64)

The "expected" column takes the result of the first cycle and scales it according to the big oh being tested -- the math for that is indicated in the sample table above. So the actual and expected for the first cycle are the same, "by definition".

It's really rare that the expected and actual match exactly, because big oh is *still* an approximation -- remember those dropped lower order terms? Also, you're running the test on a multi-tasking machine, and other things are probably going on while the test is running. But the results should be close enough to tell you it's $O(n^2)$, for example, and not one of the others.

Code For Timing Tests: O(n) And Up

Use the following code as is, adding your identifying comments and couts, and adding or modifying the **bolded** code. In each cycle, before the timing starts, declare your data structure and fill it to size n. Use **rand( )** to generate random numbers, if you wish.

```
#include <iostream> // for cout, ios, and endl
#include <string> // for string
using namespace std;

#include <cassert> // for assert
#include <cmath> // for log and pow
#include <cstdlib> // for srand and rand
#include <ctime> // for clock(), clock_t, time, and CLOCKS_PER_SEC

// your H file #include(s) go here, plus any more library includes

int main()
{
  srand(time(0)); rand(); // seed the random number generator (in case you need it)
```

```
  // programmer customizations go here
  int n = 500; // THE STARTING PROBLEM SIZE (MAX 250 MILLION)
  string bigOh = "O(n)"; // YOUR PREDICTION: O(n), O(n log n), or O(n squared)

  cout.setf(ios::fixed);
  cout.precision(4);
  double elapsedSecondsNorm = 0;
  double expectedSeconds = 0;
  for (int cycle = 0; cycle < 4; cycle++, n*= 2)
  {
    // problem setup goes here -- create a data structure of size n

    // assert that n is the size of the data structure if applicable
    //assert(a.size() == n); // or something like that...

    // start the timer, do something, and stop the timer
    clock_t startTime = clock();
    // do something where n is the "size" of the problem
    clock_t endTime = clock();

    // validation block -- assure that process worked if applicable

    // cleanup if applicable

    // compute timing results
    double elapsedSeconds = (double)(endTime - startTime) / CLOCKS_PER_SEC;
    double factor = pow(2.0, double(cycle));
    if (cycle == 0)
      elapsedSecondsNorm = elapsedSeconds;
    else if (bigOh == "O(n)")
      expectedSeconds = factor * elapsedSecondsNorm;
    else if (bigOh == "O(n log n)")
      expectedSeconds = factor * log(double(n)) / log(n / factor) * elapsedSecondsNorm;
    else if (bigOh == "O(n squared)")
      expectedSeconds = factor * factor * elapsedSecondsNorm;

    // reporting block
    cout << elapsedSeconds;
    if (cycle == 0) cout << " (expected " << bigOh << ')';
    else cout << " (expected " << expectedSeconds << ')';
    cout << " for n=" << n << endl;
} }
```

Here's the format you can expect from this code:

```
0.1436 (expected O(n)) for n=500
0.2742 (expected 0.2872) for n=1000
0.5442 (expected 0.5744) for n=2000
1.0828 (expected 1.1488) for n=4000
```

Running Timing Tests On A Multi-Tasking Computer

Note that other things are going on during your testing, if you are using Windows or OS X. Plus, we're ignoring higher order terms in determining big oh. All of the means we are not testing an ideal situation in an ideal environment. So you can expect results to vary.

As much as you can, close any other applications that may be running, while you perform your timing tests. Set the initial capacity of any data structures you might use to be large enough that capacity adjustments don't happen -- we're not timing that. And run in "release mode". All you really have to do is confirm that the results produced match *your* big oh determination better than any other possible big oh.

And don't use the results of these tests to tell you what is the big oh -- that's doing this backwards. Determine what *you* think is the big oh analytically or by reasoning, and confirm with these timing test. For very complicated f(n)'s, it's possible to use timing tests to "fit" a big oh to the results, but not by using the timing tests taught here.

An Example: Nested For-Loop Sorting

For example, you don't need an **H file** for this, but you could confirm that nested for-loop sorting is $O(n^2)$ by using this for the "**problem setup**":

```
int* a = new int[n];
for (int i = 0; i < n; i++) a[i] = rand();
```

No **assertion** is needed because there's no data structure object with a **.size( )** function to call.

For the "**do something**" part, do the sorting:

```
for (int i = 0; i < n; i++)
  for (int j = i + 1; j < n; j++)
    if (a[j] < a[i])
      swap(a[i], a[j]); // required algorithm C++ library
```

For the "**validation**":

```
for (int i = 1; i < n; i++)
  assert(a[i - 1] <= a[i]);
```

Then for "**cleanup**":

```
delete [ ] a;
```

Code For Timing Tests: O(1) and O(log n)

These two get their own section because they are so fast -- too fast to time. The solution is to do them lots of times and hope that takes enough runtime to register. Here's the modified program:

```
#include <iostream> // for cout, ios, and endl
#include <string> // for string
using namespace std;

#include <cassert> // for assert
#include <cmath> // for log and pow
#include <cstdlib> // for srand and rand
#include <ctime> // for clock(), clock_t, time, and CLOCKS_PER_SEC
```

```
// your H file #include(s) go here, plus any more library includes

int main()
{
  srand(time(0)); rand(); // seed the random number generator (in case you need it)

  // programmer customization go here
  int n = 500; // THE STARTING PROBLEM SIZE (MAX 250 MILLION)
  const int reps = n / 100; // one percent of starting n

  cout.setf(ios::fixed);
  cout.precision(4);
  double elapsedSecondsNorm = 0;
  for (int cycle = 0; cycle < 4; cycle++, n*= 2)
  {
    // problem setup goes here -- create a data structure of size n

    // assert that n is the size of the data structure if applicable
    //assert(a.size() == n); // or something like that...

    // start the timer, do something, and stop the timer
    clock_t startTime = clock();
    for (int rep = 0; rep < reps; rep++)
    {
      // do something where n is the "size" of the problem
    }
    clock_t endTime = clock();

    // validation block -- assure that process worked if applicable

    // compute timing results
    double elapsedSeconds = (double)(endTime - startTime) / CLOCKS_PER_SEC;
    double factor = pow(2.0, double(cycle));
    if (cycle == 0) elapsedSecondsNorm = elapsedSeconds;
    double expectedSecondsLog = log(double(n)) / log(n / factor) * elapsedSecondsNorm;

    // reporting block
    cout << elapsedSeconds;
    if (cycle == 0) cout << " (expected)";
    else cout << " (expected " << elapsedSecondsNorm << " to " << expectedSecondsLog << ')';
    cout << " for n=" << n << endl;
  }
}
```

There is no distinction made between O(1) and O(log n) -- the results are too close to distinguish, given their variability. But you can at least confirm that they are one or the other, and not O(n) or higher.

An Example: Stack Push

For example, using your arrayed Stack.h, do this for the "**problem setup**":

```
Stack<int> stack(n + reps); // enough capacity that doubling is not needed
for (int i = 0; i < n; i++) stack.push(rand());
```

The "**assertion**" is:

```
assert(stack.size() == n);
```

The "**do something**" is:

```
  stack.push(rand());
```

The "**validation**" is:

```
assert(stack.size() == n + reps);
```

And there is no cleanup to do.