31251 – Data Structures and Algorithms

Week 3

Luke Mathieson / Marco Tomamichel

In this week's episode:

- **Theory of Algorithms**
  - Comparing algorithms using Big-Oh notation (with many examples)
- More C++ concepts
  - `vectors`
  - Templates
  - Iterators

Analysing and Comparing Algorithms: Big-Oh Notation

- How do we reliably compare algorithms?
  - Testing with example inputs gives good information, but is limited to the cases you test.
  - How much of that information comes from the choice of computer, programming languages, test data?
- We need a way of comparing the abstract algorithms themselves.

- If we have two algorithms that solve the same problem, what things are we interested in?
  - How long they take.
  - How much memory space they take up.
- How do we know what resources an algorithm will use for any of the infinite number of possible inputs?

- Consider two functions $f(n), g(n)$ that map natural numbers onto natural numbers.
- Usually, n is the length of the input and $f(n)$ could be the maximal running time for any input of length n.
- We say $f \in O(g)$ ("f is in big-oh of g") if:

  $$\exists c \in \mathbb{R}_+, N_0 \in \mathbb{N} \text{ such that } \forall n \geq N_0, \text{ we have } f(n) \leq c \cdot g(n)$$

- Or: for big enough numbers, $f(n)$ is less than or equal to a constant times $g(n)$.
- Or:

  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

Some Examples

- $f(n) = n$, $g(n) = 2n \rightarrow f \in O(g)$.
- $f(n) = n$, $g(n) = \frac{1}{2}n^2 \rightarrow f \in O(g)$.
- $f(n) = n^2$, $g(n) = n \rightarrow f \notin O(g)$ (but $g \in O(f)$).
- $f(n) = 50n$, $g(n) = n \rightarrow f \in O(g)$ (and $g \in O(f)$).
- $f(n) = \log n$, $g(n) = n \rightarrow f \in O(g)$.
- $f(n) = n$, $g(n) = 2^n \rightarrow f \in O(g)$.
- $f(n) = 23n^3$, $g(n) = 13n^4 \rightarrow f \in O(g)$.

Proving these relationships

- These can all be proved using a variety of techniques:
  - Induction
  - Algebraically
  - Limit based definitions.
- We'll leave the proofs for now, but there's a couple of handy rules:
  - You can always ignore constant multipliers, i.e. you can treat $c \cdot f(n)$ as $f(n)$.
  - $O(\cdot)$ is transitive: if $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$.
  - $c \cdot n^k \in O(n^{k+1})$ for any c and k.
  - $\log n \in O(n)$.
  - $f(n) + g(n) + h(n) + \ldots \in O(\max\{f(n), g(n), h(n), \ldots\})$.

Some handy names

- n is "linear".
- n^2 is "quadratic".
- n^3 is "cubic".
- n^k for any k is "polynomial".
- $\log n$ is "logarithmic".
- $(\log n)^k$ for any k is "poly-logarithmic".
- k^n for any k is "exponential".

How does this help us?

- Back to the two algorithms $\mathcal{A}$ and $\mathcal{B}$:
  - If we can work out functions that describe the running time, we can now compare them and decide which is the fastest (in the long run).
  - Let n be the size of the input.
  - If the running time of $\mathcal{A}$ is $T_{\mathcal{A}}(n) = n^2$ and the running time of $\mathcal{B}$ is $T_{\mathcal{B}}(n)$ ...
  - ... we can work out that $T_{\mathcal{B}} \in O(T_{\mathcal{A}})$, so $\mathcal{B}$ must be at least as fast *asymptotically*.
  - Because $\mathcal{A}$'s running time is always at least as large (up to a constant) for large enough inputs.

- How do we get these functions then?
- In an abstract sense, running time is really the number of steps the algorithm takes for a given input size.
  - This abstracts out programming languages and computers.
- So “all” we need to do is count the number of steps.

A Simple Example

```
begin

  a = 1
  b = 2
  c = a + b

  print c

end
```

This code does the same thing for any “input” (it doesn’t really take any), so $T(n) = 4$ (ish).

A Less Simple Example

```
void printArray(int a[], size n){

  for (int i = 0; i < n; i++){

    cout << a[i];

  }

}
```

At each iteration we do a check to see if `i` is large enough to stop, print something out, and add one to `i`. We also initialise `i` once. Assuming printing is one step (is this reasonable?), $T(n) = 1 + 3n \in O(n)$.

```
for (int i = 0; i < n; i++){
  for (int j = 0; j < n; j++){
    System.out.println(i + " " + j);
  }
}
```

For each iteration of the outer loop, we have n iterations of the inner loop. For each iteration of the inner loop, we do one thing. So if the outer loop runs n times: $T(n) = n \cdot n \cdot 1 = n^2 \in O(n^2)$.

Some other properties and notations

- If $f \in O(g)$ then $g \in \Omega(f)$. $\Omega(\cdot)$ is defined the same way as $O(\cdot)$, but with $\geq$, rather than $\leq$: f is in $\Omega(g)$ if

  $$\exists c \in \mathbb{R}_+, N_0 \in \mathbb{N} \text{ such that } \forall n \geq N_0, \text{ we have } f(n) \geq c \cdot g(n)$$

- If $f \in O(g)$ and $f \in \Omega(g)$, we write $f \in \Theta(g)$ (and $g \in \Theta(f)$) — this is an equivalence relation.
- For example, $cn \in \Theta(n)$ for any c.

Some other properties and notations

- If we want to indicate that f is in $O(g)$ but not in $\Theta(g)$, we can use the "small-oh" notation.
- o is defined similarly to O, but it requires the function to grow strictly slower. In terms of limits this looks as follows: $f \in o(g)$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

- For example, $\log(n) \in o(n)$ or $n^k \in o(n^{k+1})$ but $4n \notin o(n)$ and $0.5n \notin o(n)$.
- Similarly, $\omega(\cdot)$ replaces $\Omega(\cdot)$ if we require the function to grow strictly faster.
- What do $O(1)$, $o(1)$, $\Omega(1)$, $\omega(1)$ mean?

In this week's episode:

- Theory of Algorithms
  - Comparing algorithms using Big-Oh notation (with many examples)
- **More C++ concepts**
  - `vectors`
  - Templates
  - Iterators

The `vector` class

- Arrays in C/C++ are difficult (as we saw).
- Lists provide an alternative to this, but
  - Lists usually don't have guarantees about how the memory is managed.
  - Lists also often come with more overhead.
- `vector`s are the middle ground!

- If you don't care what's inside, you can think of `vectors` as dynamically size arrays.
- Let's see one in action. (Demo.)

- What are they really? How does this sorcery work?
- `https://gcc.gnu.org/onlinedocs/gcc-4.6.3/libstdc++/api/a01116_source.html`

Templates

- Wait... what that thing in the angle brackets (<>)?
- If you've used generics in Java, this is the C++ version: templates!
- Templates provide a way to write certain types of code once:
  - If the code doesn't care about the types it's working with.
- So we can easily make `vectors` that hold different types without rewriting the code.
- They're fairly simple to work with, but provide lots of places to leave something out (and cause weird compile errors).
- For the interested, do the Challenge ”A More Advanced Linked List“ on Ed!

Iterators!

- In this week's tutorial you'll see the `tail()` function.
- It's a really annoying way to access the list right?
- What we would like is a simple way to get the next element, that's still compatible with information hiding.
- Iterators are the answer to this problem.

- Iterators in Java are fairly well put together.
  - In particular, the inheritance structure is well thought out. A little wordy, but simple.
- As you might expect, in C++ they're a lot more flexible, but you pay for this with some ugly code.
- Also, the inheritance structures are not as well defined, so using them is a little more cumbersome.

```cpp
#include<iostream>
#include<vector>


int main() {

    std::vector<int> v;

    v.resize(10);
    v.at(2) = 1;
    v[5] = 2;

    v.push_back(3);

    std::cout << "size: " << v.size() << std::endl;

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << std::endl;
    }
}
```