CSE3305 Formal Methods II

Lecture 2
Analysing Algorithms

Dr. Kevin Korb
Clayton School of Info Tech
Building 75 (STRIP), Rm 121
korb@csse.monash.edu.au
Lecture Overview

- Why Analyse?
- Importance of order
- Search and logs
- Algorithm for analysing algorithms
- $O$ Notation

**Reading:** Workbook, chapter 1

**Optional Reading:** Neapolitan & Naimipour, Chap 1 & App A
What is Computer Science?

- **Theory**
  - Limits of computation
  - Complexity: abstract comp theory; practical (analysis of algs)

- **Method**
  - Problem solving, search
  - Simulation: stochastic search; discrete-state sim; artificial life; artificial economics

- **Application**
  - Artificial intelligence, bioinformatics
  - Image processing, graphics, web
Why Analyse Algorithms?

Answer practical Q’s about computational complexity in particular:

- time
- space
- time vs. space

- Estimate resource requirements
  - time, space, other computational resources

- Find improvements (compromises)
  - Better algorithms
  - Better problems
  - More realistic goals

- Select an algorithm given a problem
Why Analyse Algorithms?

Efficiency is *always* an issue in programming!

When ignored – e.g., using rules of thumb, off-the-shelf packages (now so popular!), etc. – it is simply being shoved in the background. Without *some* analysis the penalty is unknowable!
Importance of order

Suppose alg A performs 3 basic ops per input plus overhead $c$

Suppose alg B performs 6 basic ops per input plus overhead $c'$

So, time complexity:

- $f_A(n) = 3n + c$
- $f_B(n) = 6n + c'$

Which is faster?
Importance of order

- $f_{A1}(n) = 3n + c$
- $f_{B1}(n) = 6n + c'$

Suppose instead:

- $f_{A2}(n) = 3n^3 + c$
- $f_{B2}(n) = 6n^2 + c'$

Which is faster?

<table>
<thead>
<tr>
<th>n</th>
<th>$f_{A1}$</th>
<th>$f_{B1}$</th>
<th>$f_{A2}$</th>
<th>$f_{B2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000003</td>
<td>7</td>
<td>1000003</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>1000030</td>
<td>61</td>
<td>1003000</td>
<td>301</td>
</tr>
<tr>
<td>100</td>
<td>1000300</td>
<td>601</td>
<td>4000000</td>
<td>30001</td>
</tr>
<tr>
<td>1000</td>
<td>1003000</td>
<td>6001</td>
<td>3001000000</td>
<td>3000001</td>
</tr>
<tr>
<td>10000</td>
<td>1030000</td>
<td>60001</td>
<td>3000001000000</td>
<td>300000001</td>
</tr>
</tbody>
</table>
Different Orders

1. Constant $c$

2. Logarithmic $a \log n + c$
   Common in algorithms that recursively reduce size of original problem. E.g., binary search.

3. Linear $an + c$
   E.g., sequential search.

4. Log linear $n \log n$
   E.g. Mergesort.

5. Polynomial $an^k + bn^{k'} + \cdots + c$ (Quadratic, Cubic, \ldots )
   P problems.

6. Exponential $a^{kn}$
   The “hard” combinatorial problems. NP-Hard and NP Complete problems (until you prove P=NP!).

7. Superexponential $a^{knk'n}$
   E.g., Search the space of dags with $n$ nodes.
Different Orders

Figure 1: Log, linear and n log n functions.

Figure 2: Same, with $2^n$ added.
Search Complexity

Problem:

Search for $x$ in an ordered list ($\geq$)

$L = \langle L_1, \ldots, L_n \rangle$

returning the index or nil

We all know the algorithm!

1. $i \leftarrow 1$;
2. LOOP: IF $L_i = x$ RETURN $i$;
3. INC $i$;
4. IF $i > n$ RETURN NIL;
5. GO LOOP;

What’s $O(f)$? Average case performance?
Search Complexity

- $O(f)$: Loop has $m$ ops and executes $n$ times in the worst case:
  \[ f \in O(mn) = O(n) \]

- Average case — $E[f(n)] = $ expected time on inputs of size $n$
  - If input is $U[1, n]$ then
    \[ E[f(n)] = \frac{mn}{2} \]
  - If prob $p$ of input not in list and o/w uniform, then
    \[ E[f(n)] = (1 - p) \frac{mn}{2} + pmn \]
Search Complexity

Let’s try a new algorithm:

1. $i \leftarrow 1$;
2. LOOP: IF $L_i = x$ RETURN $i$;
3. INC $i$;
4. IF $i > n$ OR $x > L_i$ RETURN NIL;
5. GO LOOP;

After all, $L$ is an ordered list. This change can only reduce the number of iterations executed, so

What is $O(f)$ now?
Search Complexity

Let’s try a new algorithm:

1. \( i ← 1; \)
2. LOOP: IF \( L_i = x \) RETURN \( i \);
3. INC \( i \);
4. IF \( i > n \) OR \( x > L_i \) RETURN NIL;
5. GO LOOP;

Afterall, \( L \) is an ordered list. This change can only improve performance, so

What is \( O(f) \) now?

Unchanged. \( O(f) \) is worst case analysis, when the input is your enemy. Average case will (normally) improve. (How?)

With an ordered list we can do better than \( O(n) \).
Search Complexity

1. first ← 1; last ← n;
2. LOOP: IF first > last RETURN NIL;
3. \( i \leftarrow \lfloor (\text{first} + \text{last})/2 \rfloor \);
4. IF \( x = L_i \) RETURN \( x \);
5. IF \( x < L_i \) THEN first ← \( i + 1 \)
ELSE last ← \( i - 1 \);
6. GO LOOP;

What is the worst case \( O(f) \)?
Search Complexity

1. $first \leftarrow 1; last \leftarrow n$;
2. LOOP: IF $first > last$ RETURN NIL;
3. $i \leftarrow \lfloor (first + last)/2 \rfloor$;
4. IF $x = L_i$ RETURN $x$;
5. IF $x < L_i$ THEN $first \leftarrow i + 1$
   ELSE $last \leftarrow i - 1$;
6. GO LOOP;

What is the worst case $O(f)$? Every comparison eliminates at least one half of $L$. In effect, we are traversing a binary tree with $n$ leaves and $n = 2^k$ levels. Therefore, $O(f) = k = \log n$.

This is typical for “divide-and-conquer” algorithms (top down recursions)
Search Complexity

At each iteration we throw out 50% of the list. Hence, we are effectively traversing a binary tree with $n$ leaves and $n = 2^k$ levels. Therefore, $O(f) = k = \log n$.

“Logs come from trees”
Algorithm to Analyse Algorithms

1. Select the algorithm to analyse

2. Characterize the input size $n$
   NB: In $f(n)$ $n$ is not the input, but the size of the input; if input is a number $a$ then $n = \log a$.

3. Identify fixed overhead cost $c$

4. Identify iterative body cost $m$

5. Determine number of iterations as a function of $n$
   (usually worst case number)
   E.g., $m^2 n^2 + c$, $mn^2 + c$, $m \log n$, ...
Order and \( O \) Notation

- \( O(f) \): functions that grow no faster than \( f \)
  Worst case analysis; upper bounds

- \( \Omega(f) \): functions that grow at least as fast as \( f \)
  Best case analysis; lower bounds

- \( \Theta(f) \): functions that grow at the same rate as \( f \)
  Describing \( f \): upper and lower bounds
Order and $O$ Notation

An alternative vision of orders:

The point is that if some function is order $O(f)$, then so too is every slower growing function; if some function is order $\Omega(f)$, then so to is every more faster function. In particular, every function is $\Omega(1)$.
Complexity Analysis

There are three main kinds of complexity analysis:

1. Worst case analysis
   Useful for avoiding catastrophe. Variations: best case, etc.

2. Average (mean) case analysis
   E.g., \( \text{Ave}(g(n)) = E[g(n)] = \sum_i P(i)g(i) \), involving a formal analysis of \( P(i) \), which is often hard.

3. Empirical profiling, e.g., sampling inputs to construct a histogram estimate of \( g(n) \). A useful fall-back, when the first two are too hard. (Useful for sanity checks!)

We emphasize the first, but will look at a few examples of the other two as well.

**Exercise 1** What is the relation between average complexity and \( \Theta \)? If they are the same, prove it; if they are different, illustrate the difference.
Complexity Theory

What is meant by (computational) complexity theory? Either

- Theoretical analysis of time (space) complexity, in $O$ notation, etc.

- Or, Supertheoretical:
  Theory of polynomial time (space) complexity (P), exponential time (space) complexity (EXP) and non-deterministic polynomial time (space) complexity (NP); limits of computation = non-computable functions

There’s a relation between the two, as we will see late in semester.
**O Notation**

**Notation:** \( \{0\} \cup \mathbb{R}^+ = \mathbb{R}^* \), the non-negative real numbers. We assume non-negative \( f \) and \( g \).

**Criterion 1 (O(f))**

\[
g \in O(f) \text{ if } \lim_{n \to \infty} \frac{g(n)}{f(n)} < \infty
\]

This says that in the limit \( g \) may vanish as a proportion of \( f \); or, it may reach a constant proportion. What it can’t do is continue growing faster than \( f \) indefinitely.

Roughly equivalent:

- ratio of growth \( g : f \) is bounded by a constant
- “\( g \) is O of \( f \)”
- \( g \in O(f) \) (note: \( O(f) \) is formally a set!)
- \( g \) is bounded by constant multiplier of \( f \)
- \( g \) grows no faster than \( f \) (eventually)
O Notation Caveats

1. This is a limit property; we don’t care how \( g \) behaves at “small” \((\leq n')\) input sizes!

2. These are criteria, not definitions. The exact definitions will appear in the workbook (chap 2?), but you will not have to work with them in this subject.
O Notation

Intent: O notation (despite appearances) is simple(r)!

Idea:

You have some messy $g$;
find a simpler $f$ that bounds it, i.e., $g \in O(f)$.

Example 1

$$g(n) = 6n^3$$

We can find a simpler $f$ such that $g \in O(f)$.

$$\lim_{n \to \infty} \frac{6n^3}{n^3} = 6 < \infty$$

Done: $g \in O(f)$.

— i.e., $g \in O(n^3)$. 

Formal Methods II
**O Notation**

**Example 2**

\[ g(n) = n^3 + n \]

We can show that \( g \in O(n^3) \), losing the minor term \( n \).

\[
\lim_{n \to \infty} \frac{n^3 + n}{n^3} = \lim_{n \to \infty} \frac{3n^2 + 1}{3n^2} = 1 < \infty
\]

by L’Hospital’s Rule.

Note that we lose irrelevant details (constant and minor factors), including machine dependencies.
**O Notation**

**Criterion 2** ($\Omega(f)$)

\[ g \in \Omega(f) \text{ if } \lim_{n \to \infty} \frac{g(n)}{f(n)} > 0 \]

In the limit $g$ grows at least as fast as $f$. Its ratio to $f$ may be unbounded (so, $f$ as a proportion of $g$ vanishes); or it may asymptote to some positive number.

Roughly equivalent:

- ratio of growth $f : g$ is bounded by a constant
- “$g$ is $\Omega$ of $f$”; $g \in \Omega(f)$
- $g$ is bounded below by constant multiplier of $f$
- $g$ always grows faster than $f$ (eventually)

The trivial $\Omega$: $\Omega(1)$
\textbf{O Notation}

\textbf{Criterion 3 (Θ(\(f\)) )}

\[ g \in \Theta(f) \text{ if } 0 < \lim_{n \to \infty} \frac{g(n)}{f(n)} < \infty \]

\(g\) is order \(f\); this time \(g\) is not allowed to vanish as a proportion of \(f\). If it did it could not serve as a bound on \(f\); since \(Θ\) implies a bound in both directions, that wouldn’t do.

- \(g\) is order \(f\)
- \(g \in O(f)\) and \(g \in \Omega(f)\)
O Notation

Criterion 4 (Little \( o(f) \))

\[ g \in o(f) \text{ if } \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0 \]

\( g \) is strictly less than \( f \) in the limit: it vanishes.