COMP3121 Notes

1 Week 1

1.1 Intro to Algorithms

What is an Algorithm?
A collection of precisely defined steps that can be executed mechanically (without intelligent decision-making).

Sequential Deterministic Algorithms:
Algorithms are given as sequences of steps, thus assuming that only one step can be executed at a given time.

Example: Two Thieves
Alice and Bob have robbed a warehouse and have to split a pile of items without price tags on them. Design an algorithm to split the pile so that each thief **believes** that they have got at least half the loop.

Solution:

Alice splits the pile in two parts, so that she believes that both parts are equal
Bob then picks the part that he believes is no worse than the other

Example: Three Thieves
Alice, Bob and Carol have robbed a warehouse and have to split a pile of items without price tags on them. How do they do this in a way that ensures that each thief **believes** they have gotten at least one third of the loot.

Solution:

Alice makes a pile of $\frac{1}{3}$ called X
if Bob agrees $X \leq \frac{1}{3}$ **then**
  Bob agrees to split the remainder with Carol
  if Carol agrees $X \leq \frac{1}{3}$ **then**
    Bob and Carol split the rest
  else
    Alice and Bob split the rest
  end if
else
  Bob reduces pile until he thinks $X \leq \frac{1}{3}$ and Alice and Carol split the rest
end if

When are proofs necessary?
We use proofs in circumstances where it is not clear that an algorithm truly does its job.

Proofs should <u>not be used to prove the obvious.</u>

1.2 Complexity

Rates of Growth

When trying to determine whether one algorithm is faster than another, we talk in terms of *asymptotics*, being long-run behavior.
- e.g if the size of the input doubles, does the function's value double?

We want to categorise the runtime performance of an algorithm by its *asymptotic rate of growth*.

Big-O Notation

Definition:
We say $f(n) = O(g(n))$ if for *large enough* n is *at most a constant multiple of* $g(n)$.

- $g(n)$ is an *asymptotic upper bound* for $f(n)$
- The rate of growth of function f is no greater than that of function g
- An algorithm whose running time is $f(n)$ scales *at least as well* as one whose running time is $g(n)$

Example
Let $f(n) = 100n$. Then $f(n) = O(n)$, because $f(n)$ is at most 100 times n for large n.

Big-Omega Notation

Definition
We say $f(n) = \Omega(g(n))$ if for *large enough* n, $f(n)$ is *at least* a constant multiple of $g(n)$.

- $g(n)$ is said to be an *asymptotic lower bound* for $f(n)$.
- Meaning the true rate of growth of function f is no less than that of function g.
- An algorithm whose running time is $f(n)$ *scales at least as badly* as $g(n)$.

Big-Theta Notation

Definition We say $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

- $f(n)$ and $g(n)$ are said to have the same asymptotic growth.
- An algorithm whose running time is $f(n)$ scales as well as $g(n)$.

Sum Property

Fact
If $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then $f_1 + f_2 = O(g_1 + g_2)$.

- This property justifies ignoring non-dominant terms.
- If f_2 has a lower asymptotic bound than f_1 then the bound on f_1 also applies to $f_1 + f_2$
- For example, if f_2 is linear but f_1 is quadratic, then $f_1 + f_2$ is quadratic.
- Especially relevant when dealing with algorithms that have two or more *sequentially executed stages.*

Product Property

Fact
If $f_1 = O(g_1)$ and $f_2 = O(g_2)$ then $f_1 \cdot f_2 = O(g_1 \cdot g_2)$.

- Especially relevant when dealing with algorithms that have two or more *nested stages.*

1.3 Logarithms

Definition
For $a, b > 0$ and $a \neq 1$, let $n = log_a b$ if $a^n = b$.

Properties

$$a^{log_a n} = n$$

$$log_a(mn) = log_a m + log_a n$$

$$log_a(n^k) = k \cdot log_a n$$

Change of Base Rule

Theorem
For $a, b, x > 0$ and $a, b \neq 1$, we have

$$log_a x = \frac{log_b x}{log_b a}$$

- The denominator is constant with respect to x!

1.4 Data Structures

Hash Tables
- Stores values indexed by keys.
- Hash functions map keys to indices in a fixed size table.
- Ideally, no two keys map to the same index, although impossible to guarantee this.
- A situation where two (or more) keys have the same hash is called a *collision.*
- There are methods to resolve collisions (e.g separate chaining, open addressing, etc)

Operations (expected)
- Search for the value associated to a given key: $O(1)$
- Update the value associated to a given key: $O(1)$
- Insert/delete: $O(1)$

Operations (worst case)
- Search for the value associated to a given key: $O(n)$
- Update the value associated to a given key: $O(n)$
- Insert/delete: $O(n)$

Binary Search Trees

...to be continued