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 not be used to prove the obvious.

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
BSTs store keys or key-value pairs in a tree, where each node has at most two children, designated at left and right.

Each node's key compares greater than all keys in its left subtree, and less than all keys in its right subtree.

Operations
Let h be the height of the tree, that is, the length of the longest path from the root to the leaf.
- Search: $O(h)$
- Insert/delete: $O(h)$

Self-Balancing Binary Search Trees
- Best Case: $h \approx log_2 n$, which is said to be balanced.
- Worst Case: $h \approx n$, which is if keys are inserted in increasing order.

All examples of Self-Balancing BSTs perform rotations to maintain certain invariants, in order to guarantee that $h = O(\log n)$ and therefore that all tree operations run in $O(\log n)$.

Binary Heaps
- Store items in a complete binary tree, with every parent comparing $\geq$ all its children for max heap.
- For min heap, it's $\leq$ instead.
- Commonly used to implement a priority queue.

Operations
- Build Heap: $O(n)$
- Find Maximum: $O(1)$
- Delete Maximum: $O(\log n)$
- Insert: $O(\log n)$

1.5 Binary Search

if the array has no elements (if r is not $\geq$ length) **then**
  x cannot be found.
else
    $m = \frac{l+r}{2}$ (midpoint of the subarray)**if** $A[m] = x$ **then**
    Occurrence found, exit the recursion $A[m] > x$
    Recurse on the left subarray A[l..m - 1]
  else
    Recurse on the right subarray A[m + 1..r]
  end if
end if

Complexity
- Worst Case: $O(\log n)$
At each step, the search space halves, which can only happen $log_2 n$ times.

Decision and Optimisation Problems

Decision problems are of the form:
Given some parameters including X, can you...

Optimisation problems are of the form:
What is the smallest X for which you can...

An optimisation problem is typically much harder than the corresponding decision problem, because there are more choices.

Discrete Binary Search
A technique of binary searching, where you find the smallest X such that $f(X) = 1$ using binary search.

Overhead is just a factor of $O(\log A)$ where A is the range of possible answers.

1.6 Sorting

Merge Sort

Algorithm
1. If $n = 1$, do nothing. Otherwise let $m = (n + 1)/2$.
2. Apply merge sort recursively to $A[1..m]$ and $A[m + 1..n]$.
3. Merge $A[1..m]$ and $A[m + 1..n]$ into $A[1..n]$.

Complexity
- Best Case: $O(n \log n)$
- Worst Case: $O(n \log n)$
- Space: $O(n)$

- Reliably fast for large arrays.
- Space requirement is a drawback.

Heapsort

Algorithm
1. Construct a min heap from the elements of A.
2, Write the top element of the heap to $A[1]$ and pop it from the heap.
3. Repeat the previous step until the array is empty.

Complexity
- Best Case: $O(n \log n)$
- Worst Case: $O(n \log n)$
- Selection sort but with selection in $O(\log n)$ rather than $O(n)$

- Reliably fast for large arrays.
- No additional space is required.
- Constant factor is larger than other fast sorts, so used less in practice.

Quicksort

Algorithm
1. Designate the first element as the *pivot*.
2. Rearrange the array so that all smaller elements are to the left of the pivot, and all larger elements are to its right.
3. Recurse on the subarrays left and right of the pivot.

Complexity
- Rearranging and partitioning the array can be done in $O(n)$.
- If the pivot is the median, best case is $O(n \log n)$.
- Worst case is $O(n^2)$

- Because the worst case is so rare, quicksort is widely used.
- Quicksort forms the basis of the default sort in many languages.

Comparison Sorting
Any comparison sort must perform $\Omega(n \log n)$ comparisons in the worst case.

Proof
- There are $n!$ permutations of the array.
- In the worst case, only one of these is the correct sorted order, and our sorting algorithm must find which permutation this is.
- An algorithm that performs k comparisons can get 2^k different combinations of results from these comparisons, and therefore can distinguish between at most 2^k permutations.
- We need to perform number of comparisons k such that $2^k \geq n!$.
- We can conclude that $k = \omega(n \log n)$ in the worst case.

Counting Sort

Algorithm
1. Create another array B of size k to store the count of each value, initially all zeros.
2. Iterate through A. At each index i, record one more instance of the value $A[i]$ by incrementing $B[A[i]]$.
3. Write $B[1]$ many ones, then $B[2]$ many twos... into A, from left to right.

Complexity
- Initialising B takes $O(k)$ time.
- Iterating through A takes $O(n)$ time.
- Final step takes $O(max(n,k))$ time... $O(n+k)$.
- Total complexity is $O(n+k)$.
- $O(k)$ space required.

Bucket Sort

Algorithm
1. Distribute the items into buckets $1, ...k$.
2. Sort within each bucket.
3. Concatenate the sorted buckets.

Now to sort each bucket using a bucket sort...

MSD Radix Sort

Algorithm
Bucket the keys by their first symbol, and recursively apply the same algorithm to each bucket.

Complexity
- There are k levels of recursion.
- In each level, there are n keys to be bucketed, in constant time.
- Worst case complexity is thus $O(nk)$.

LSD Radix Sort

Algorithm
Bucket the keys by their last symbol, then sort all keys by their second last symbol, etc.

Complexity
- Worst case complexity is again $O(nk)$.

1.7 Graphs

A graph is a pair (V, E) where V is the *vertex set* and E is the *edge set*, where each edge connects a pair of vertices.

- We refer to V as vertices.
- And E as number of edges.

Adjacency List
- For each vertex v, stores a list of edges from v.

Operations
- Test for edge from v to u: $O(deg(v))$
- Iterate over neighbours of v: $O(deg(v))$

Adjacency Matrix
- Store a matrix, where each cell stores information about the edge from u to v or lack thereof.

Operations
- Test for edge from v to u: $O(1)$
- Iterate over neighbours of v: $O(V)$

Depth-First Search
From a vertex v:
- mark v as visited and,
- recurse on each unvisited neighbor of v.
- time complexity is $O(V + E)$ using a stack.

Breadth-First Search
From a vertex v:
- mark v as visited and,
- mark each unvisited neighbour of v as visited,
- mark each of their unvisited neighbors as visited, etc.
- time complexity is $O(V + E)$ using a queue.

Trees

Definitions
- An undirected graph is *connected* if every pair of vertices can reach each other by a sequence of one or more edges.

A tree is a *connected graph* because:
- there is a unique simple path between each pair of vertices.
- there is one fewer edge than vertices, or
- there are no cycles, or
- the removal of any edge disconnects from the graph.