School of Computer Science

# COMP SCI 1103/2103 Algorithm Design & Data Structure

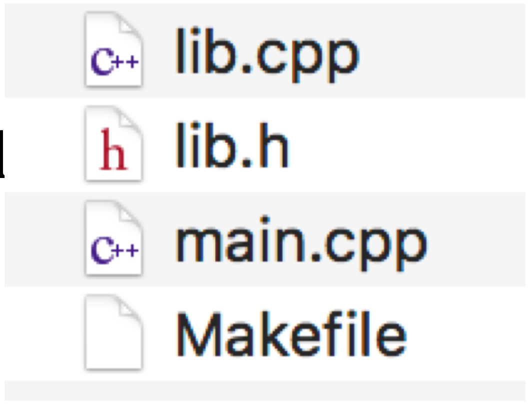
## Complexity - Bounds and Notation

## Review

- Makefile
- The number of comparisons we need when we do an exhaustive search on a sorted array
  - We got the idea that binary search is better
  - But we still do not know how better!

# What does a compiler do?

- g++ -c main.cpp lib.cpp
  - Makes two object files
- What if a cpp files includes lib.h as well as something else that includes lib.h?
  - `#include "lib.h"`
  - Copy the whole header there
  - Should be declared once; If you are including that in more than one file use `ifndef`
- g++ command also **links** the object files and makes an exe
- Is it allowed to have one declaration repeated in two object files?
  - How about the definition of that?



```
#ifndef __LIB_H__
#define __LIB_H__


//The declaration of your class/library


#endif
```

## Overview

- Today we will talk about complexity
- Definitions
  - Big O

## Efficiency

- Input size plays a key role! (what is input size in the searching example?)
- Structures that we use in our algorithm matter!
  - Compare adding a new element to the beginning of an array and a linked list!
  - This means that design matters!
- Make sure we analyze based on the problem assumptions
  - What if the given array in previous example was not sorted?!
- Now! how to really express the efficiency of an algorithm?

## What should we do?

- How long will it take to run a given algorithm?
  - Compare algorithms. Not machines.
  - An algorithm can be implemented by any language
  - Just consider a machine model and count its basic operations
- Input size matters. Large input size is important.
  - We want to find the complexity in asymptotic sense.
- Read details in the following 5 slides

## Model

- One application, different running times on different computers
  - Not all computers are the same.
  - We talk about complexity as if it can tell us how long something will take
- We are looking for comparative measures, to be able to rank algorithms.
- Many of the complexity classes that we use have been developed on machine models, not actual computers
- We simply count the number of basic operations that needs to be done by the machine
  - Each basic operation needs a constant time, independent of the size of the input

## Complexity

- Given that we would like to have efficient mechanisms, we need to be able to compare the relative efficiency of our solutions (algorithms).
- The measure of complexity for an algorithm gives us an indication of how much time, or space, it will consume, given the size of its input.
- This is a fundamental concept in computer science.

## Assumptions

- Large inputs are important
- Algorithm
  - An algorithm is a clearly specified set of instructions to be followed to solve a problem. (explained in pseudo-code or text)
  - Can be implemented using any language
- Run an algorithm on a machine model
  - Not a specific computer
  - We assume that each basic operation takes one unit of time

## Size of the input

- What do we mean by the size of the input?
- If we are searching through books for a certain book, the size of the input is the **NUMBER** of books.
  - The more books we have, the longer the worst-case search.
- In runtime analysis, we don’t care about small input size.

## Size matters

- If we were only ever looking at small problems, inefficiency wouldn't matter.
- If you are looking for your socks in your suitcase, this is a small search space and an easily defined condition.
  - Who cares about efficiency here! Unless you are doing this simple job for a very large number of times
- What if you were looking for any piece of clothing that fitted you, in a space the size of Adelaide?
  - The significance of efficiency can be sensed here!

## Exhaustive Search

- An exhaustive search is one where you look at absolutely everything.
- We may have some kind of stopping condition, but the worst case, if what we're looking for isn't there, is that we have to look at everything.
- Consider an algorithm that is picking an item out of n, uniformly at random, at each iteration. Is this an exhaustive search?
  - no
- Exhaustive search still needs to be a *systematic* search.
- Often called a *brute force* search.

## Why systematic?

- If what you are looking for, let's say 1 object, is in a pile of n other objects, you have a 1/n chance of picking out that object by chance, at each selection.
- If, in a systematic search, you reduce the number of objects left to search, then your chance steadily increases until either:
  - you find it
  - you get to n=1 and it's not there.

## The Ideal Case

- We look for algorithms that take as little time and space as possible for large inputs.
- We would love to have algorithms that take constant time, or use roughly the same space, regardless of the size of the input.
- Some of these exist, but they don't exist for most problems.
- Can we find/ do we need exact number of steps?
- The growth rate matters
  - How do you compare c.n to n.n
- Do some analysis before implementation.

## Back to my inefficient search algorithm!

The computational complexity of one run of this algorithm will be somewhere between the worst case and the best case.
Let us draw this with respect to n.

## Upper and lower bounds

- I'm talking about my teacher with you!
- You want me to describe him! You ask how old he is.
- I don't know! But I can give a range!
  - Between 0 and 122!
  - Between 35 and 40
- Note that the first one is also correct! But useless.

## Complexity Bounds

- In many cases, we can not tell the precise running time complexity of an algorithm.
- We are interested in average or worst case scenarios
- If we can define the worst case complexity of the algorithms in terms of:
  - a lower bound (**I have found a worst case** that takes **at least** this much!)
  - an upper bound (this is as bad as it can get in general. I'm sure that **there exists no cases that take worse than this**.)
- then we can compare algorithms, if our bounds are good enough (e.g. if the bounds are tight).

## Example

- Suppose that we want to search for an item over every element in a list that is n elements long. That will take n operations in worst case.
- What about a list that is twice as long?
  - Again size of input. New n
- Is n a lower bound on the worst case complexity?
- Is log n a lower bound?
- Is n^2 a lower bound?
- Is n an upper bound?
- Is log n an upper bound?
- Is n^2 an upper bound?

## Best case

- You have an array which contains sorted numbers: 1, 2, 3, … , 100.
- If you start searching at the start, looking for 1, you will immediately find it.
- This is the best case situation - either you have very little to deal with or you are just lucky with your first operation.
- Pretty useless! Don’t confuse yourself with best cases. Don’t mix it with lower bounds! When we find lower bounds we usually find it for worst case or average case. It is also possible to find a lower bound on the best case, but that is not useful in the literature.

## Worst case

- Now imagine the same array, but you start at the front and search through looking for 100.
- Now you will have to search through all of the elements to get to the last one.
- This, or the situation where what you are looking for something that is not in the list, is the worse case.
- What is the worst case complexity for an array of size n?
  - n time units
- This means that *n time units*, is a lower bound for time complexity of our algorithm

## Comparing algorithms

- We know that we can’t predict exactly how long an algorithm will take to run.
- If we can estimate it, reliably, in terms of some reference point, we can compare algorithms with each other.
- But we need to agree on these estimates.
  – We define the complexity of algorithms in terms of their behaviour as n gets large, in the worst case scenario.
  – However, in many situations the average case is important, which is harder to analyze.

## Questions!

- As a customer who wants to decide about buying an algorithm,
  - Is the complexity of the algorithm important in the worst case?
  - Is the complexity of the algorithm important in average case?
  - Is the complexity of the algorithm important in the best case?
- We focus on the worst case
- As a customer
  - Is an upper bound on the worst case important?
  - Is a lower bound on the worst case important?

## Questions!

- As the algorithm designer
  - Do you try to develop an algorithm with a high complexity or low?
  - Can you always tell the precise complexity of your algorithm?
  - Are you happy if an analyst finds a high upper bound or low?
  - Are you happy if an analyst finds a high lower bound on the worst case, or low?
  - Are you happy if an analyst finds high/low upper/lower bounds for your opponent's algorithm?

- As an analyst
  - Do you try to find a high upper bound or low?
  - Do you try to find a high lower bound or low?
  - How can you prove that an upper bound is tight?
  - How can you prove that a lower bound on the worst case is tight?

## Summary

- The efficiency of your code is related to the structures AND algorithms that you use.
  - As we'll see, these are heavily bound together.
- You have to understand why one algorithm is better than another and why one structure is a better choice at certain times.
- Understand complexity and we can pick wisely.
- Use upper and lower complexity bounds
- Best-case is pretty useless. What matters is worst-case and average-case complexity