School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Complexity - Bounds and Notation

Overview

- Today we will talk about efficiency of running time and memory use in algorithms
- Classifying problems based on their running time/memory use.

Design affects efficiency

- The choices we make in our design affect run time and memory use of our programs
  - Consider our use of memoisation and tabulation. Will this always be faster? Can you think of cases when it won't?
  - How about memory use?
- Memory use and running time can often be traded off for one another (reducing run time by using more memory or vice versa)
- But not designing with efficiency in mind can waste both

Efficiency

- The size of the data the algorithm has to work on plays a key role.
- Structures that we use in our algorithm matter.
- Make sure we analyze based on the problem assumptions
  - Consider searching for a value in an array
    - What if the given array is sorted?
    - What if it's not (should we sort it?)

How should we measure efficiency?

- How long will it take to run a given algorithm?
- Can this question be answered in general? How long does it take to run our recursive Fibonacci program?

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 programs, 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 are concerned with how much the running time increases as the input size increases (it's growth rate).

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 – *almost* the least efficient approach

- 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 Algorithm

- 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 many problems.
- Do some analysis before implementation to see if you can find a more efficient algorithm.

An inefficient search algorithm

Consider the problem of searching for a value in a sorted list.

We decide to code this search as:

```
for(i=0; i<sizeOfList; i++) {
   if (list[i] == value)
        // found it!!
}
```

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

- We'd like the bounds to be as tight as possible. To be useful.
- Example:
  - 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 upper and lower bound is also correct. But not as useful.

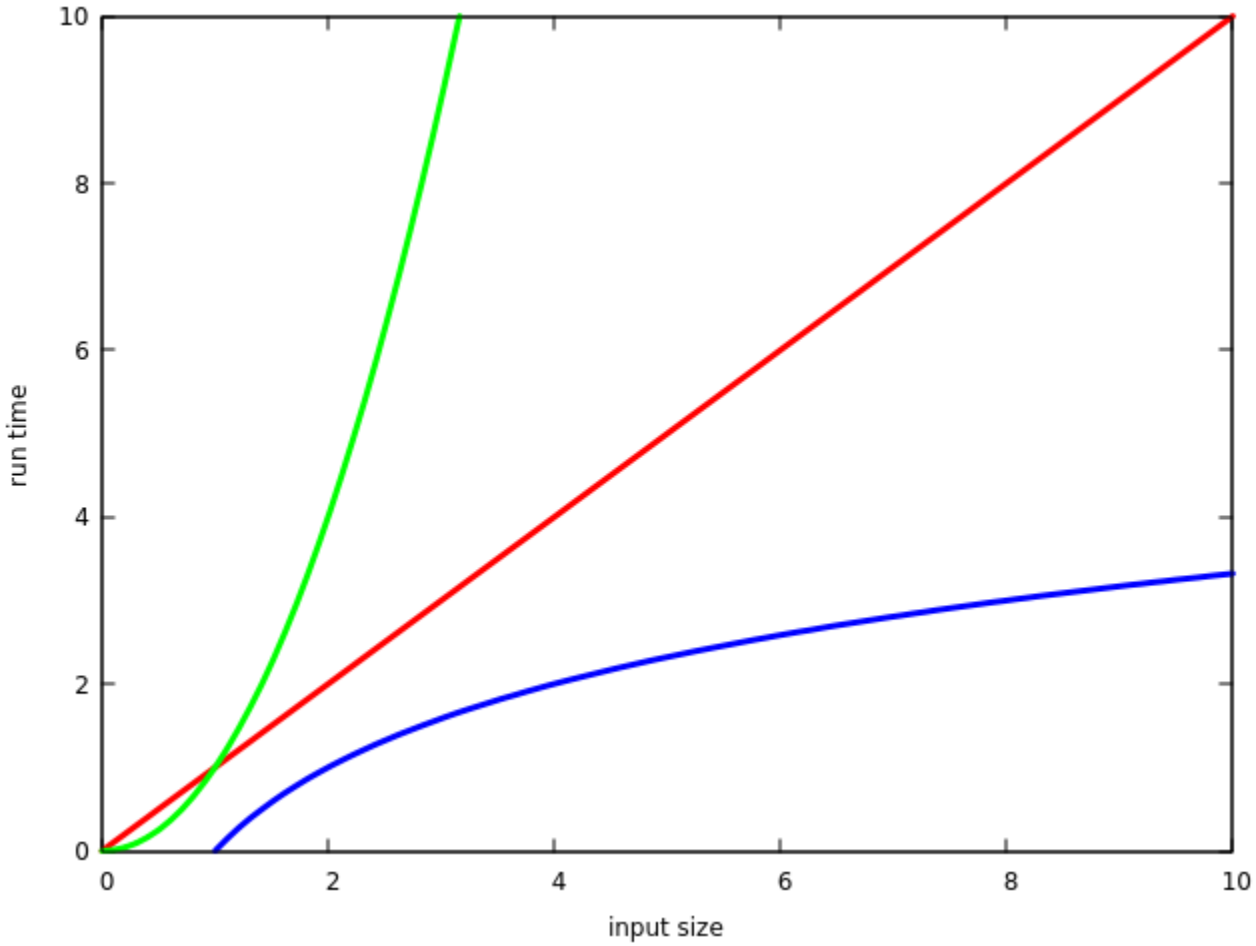
Complexity Bounds

- In many cases, we cannot 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?
- 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?

Input vs Run-time



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 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 often useful.

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?
- 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.

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


THE UNIVERSITY
of ADELAIDE