Analysing Algorithms

Part I

Roadmap

- Features of an algorithm
- Algorithmic efficiency
- Performance & complexity
- Orders of magnitude & complexity
- Best, average & worst cases

Recap

- An algorithm is a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer (Oxford English Dictionary)
- Algorithms can be thought of like a “recipe” or set of instructions to be followed to achieve the desired outcome
- Many different algorithms could be designed to achieve a similar task
  - What criteria should we take into account?
  - How should we compare different algorithms, and decide which is most appropriate for our use case?

Features of a well-designed algorithm

- **Input**: An algorithm has zero or more well-defined inputs, i.e data which are given to it initially before the algorithm begins
- **Output**: An algorithm has one or more well-defined outputs i.e data which is produced after the algorithm has completed its task
- **Finiteness**: An algorithm must always terminate after a finite number of steps
- **Unambiguous**: Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case
- **Correctness**: The algorithm should consistently provide a correct solution (or a solution which is within an acceptable margin of error)
- **Feasibility**: It should be feasible to execute the algorithm using the available computational resources
- **Efficiency**: The algorithm should complete its task in an acceptable amount of time

Efficiency

- Different algorithms have varying space and time efficiency
  - E.g. there are several commonly used sorting algorithms, each with varying levels of efficiency
  - All algorithms are not created equal!
- Time efficiency considers the time or number of operations required for the computer takes to run a program (or algorithm in our case)
- Space efficiency considers the amount of memory or storage the computer needs to run a program/algorithm
- In this module we will focus on time efficiency

Analysing efficiency

- There are two options for analysing algorithmic efficiency:
  - **A priori analysis** – Evaluating efficiency from a theoretical perspective. This type of analysis removes the effect of implementation details (e.g. processor/system architecture). The relative efficiency of algorithms is analysed by comparing their order of growth. Measure of complexity.
  - **A posteriori analysis** – Evaluating efficiency empirically. Algorithms which are to be compared are implemented and run on a target platform. The relative efficiency of algorithms is analysed by comparing actual measurements collected during experimentation. Measure of performance.

Complexity

- In general, complexity measures an algorithm's efficiency with respect to internal factors, such as the time needed to run an algorithm
- External Factors (not related to complexity):
  - Size of the input of the algorithm
  - Speed of the computer
  - Quality of the compiler

Performance vs. complexity

It is important to differentiate between:

- **Performance:** how much time/memory/disk/... is actually used when a program is run. This depends on the computer, compiler, etc. as well as the code
- **Complexity:** how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved or input dataset gets larger?

Note that complexity affects performance but not the other way around

Performance vs. complexity

- Note that algorithms are **platform independent**
- i.e. any algorithm can be implemented in an arbitrary programming language on an arbitrary computer running an arbitrary operating system
- Therefore, empirical comparisons of algorithm complexity are of limited use if we wish to draw general conclusions about the relative performance of different algorithms, as the results obtained are highly dependent on the specific platform which is used
- We need a way to compare the complexity of algorithms that is also platform independent
- Can analyse complexity mathematically

Comparing complexity

- We can compare algorithms by evaluating their running time complexity on input data of size n
- Standard methodology developed over the past half-century for comparing algorithms
- Can determine which algorithms scale well to solve problems of a nontrivial size, by evaluating the complexity the algorithm in relation to the size n of the provided input
- Typically, algorithmic complexity falls into one of a number families (i.e. the growth in its execution time with respect to increasing input size n is of a certain order)
- Memory or storage requirements of an algorithm could also be evaluated in this manner

Orders of magnitude

- **Order of Magnitude** is a mathematical term which basically tells the difference between classes of numbers
- Think of the difference between a room with 10 people, and the same room with 100 people. These are different orders of magnitude
- However, the difference between a room with 100 people and the same room with 101 people is barely noticeable. These are of the same order of magnitude

Orders of Magnitude

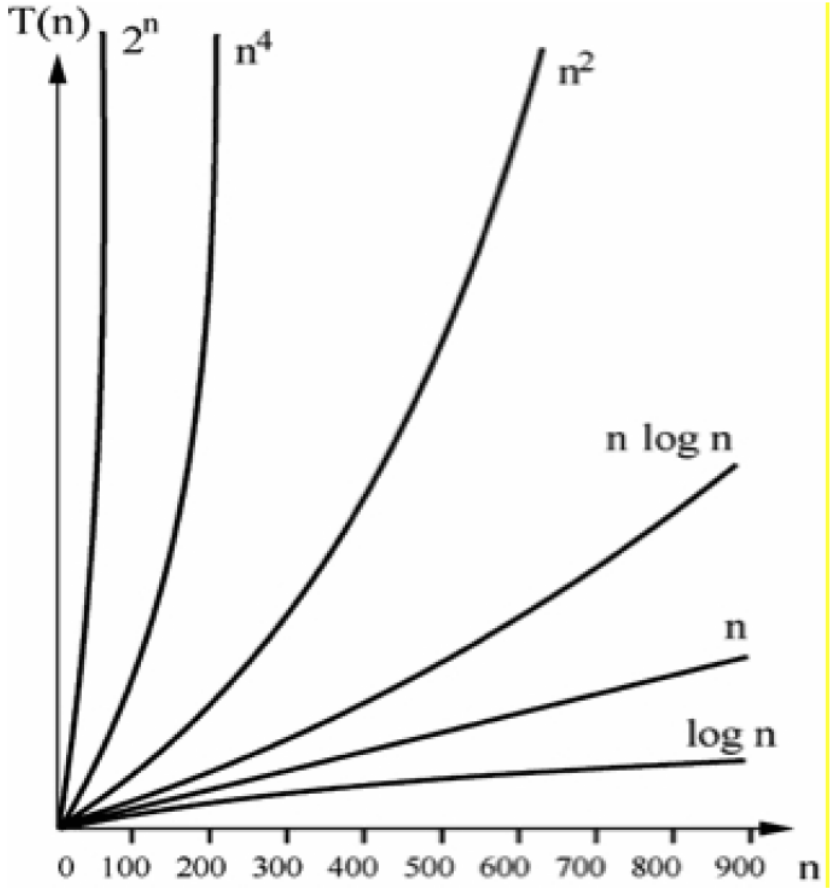
- How many gumballs in this gumball machine?
- If we guess 200, that's probably pretty close
- Whereas a guess of 10,000 would be way off

Complexity families

- We will use the following classifications for order of growth (listed by decreasing efficiency, with the most efficient at the top):
  - Constant
  - Logarithmic ($\log(n)$)
  - Sublinear
  - Linear (n)
  - $n \log(n)$
  - Polynomial (e.g. n^2, n^3, n^4, etc.)
  - Exponential

Typical complexity curves

Running time $T(n)$ is proportional to:	Complexity:
$T(n) \propto \log n$	logarithmic
$T(n) \propto n$	linear
$T(n) \propto n \log n$	linearithmic
$T(n) \propto n^2$	quadratic
$T(n) \propto n^3$	cubic
$T(n) \propto n^k$	polynomial
$T(n) \propto 2^n$	exponential
$T(n) \propto k^n;\ k > 1$	exponential



Evaluating complexity

- When evaluating the complexity of an algorithm, keep in mind that you must identify the most expensive computation within an algorithm to determine its classification
- For example, consider an algorithm that is subdivided into two tasks, a task classified as linear followed by a task classified as quadratic. The overall performance of the algorithm must therefore be classified as quadratic

Best, average and worst cases

- As well as the size n of the input, the characteristics of the data in the input set may also have an effect on the time which an algorithm takes to run
- There could be many, many instances of size n which would be valid as input; it may be possible to group these instances into classes with broadly similar features
- Some algorithm “A” may be most efficient overall when solving a given problem. However, it is possible that another algorithm “B” may in fact outperform “A” when solving particular instances of the same problem
- The conclusion to draw is that for many problems, no single algorithm exists which is optimal for every possible input
- Therefore, choosing an algorithm depends on understanding the problem being solved and the underlying probability distribution of the instances likely to be treated, as well as the behaviour of the algorithms being considered.

Best, average and worst cases

- **Worst case**: Defines a class of input instances for which an algorithm exhibits its worst runtime behaviour. Instead of trying to identify the specific input, algorithm designers typically describe properties of the input that prevent an algorithm from running efficiently.
- **Average case**: Defines the expected behaviour when executing the algorithm on random input instances. While some input problems will require greater time to complete because of some special cases, the vast majority of input problems will not. This measure describes the expectation an average user of the algorithm should have.
- **Best case**: Defines a class of input instances for which an algorithm exhibits its best runtime behaviour. For these input instances, the algorithm does the least work. In reality, the best case rarely occurs.