Computing Concepts and Paradigms

Lecture 6
Efficiency

- While the speed of a computer might help in solving problems a bit faster, it is irrelevant to algorithmic efficiency.
- There is no computer today which can solve the Travelling Salesman Problem of 100 cities in less than millions of years.
- Likewise no computer is able to factor integers of around 300 digits in less than millions of years.
- Likewise for the towers of hanoi problem for number of discs larger than 64.
- All these problems will not be solved faster by faster computers but only by innovative algorithmic design.
Optimisation through program transformations

• Many compilers today are able to make code more efficient by converting pieces of code into code that runs faster, yet still with the same semantic behaviour.

• For example a value which is calculated several times can be transformed to a single calculation:
  ex: for i=1 to N
      output (i+(100/20))
  will be converted to
      temp = 100/20;
      for i=1 to N
          output (i+(temp))

• These optimisations make part of compiling theory which is quite advanced, yet is still not definite.
Time Complexity

• When we refer to how long an algorithm takes, instead of referring to time we tend to refer to a measure $O$ (big-O) which give us the order of runs (or comparisons) needed depending on the size of an input.

• For example: searching for a number in an unsorted list of size $N$ takes at worst $N$ checks. Thus we say the algorithm runs in order $N$, i.e. $O(N)$.

• Note that to calculate the time taken on a specific computer we merely use $k\cdot O(N)$ where $k$ would be the amount of time to perform one iteration on a specific computer.

• N.B. Having a computer which is twice as fast as another computer will not really matter when $N$ is large. Example $k\cdot O(N!) \approx (k/2)\cdot O(N!)$ for large $N$.
Time Complexity (cont)

• In addition the constant terms inside the big O notation are ignored since it’s the relationship to the input size which matters for $O()$.
  
  example $O(N/2) = O(N)$

• Note that searching in an unsorted list has **average** search time $O(N/2)$ yet we still quote the algorithm as $O(N)$.

• Also we are just measuring the comparisons, not the other instructions which come after and before in the algorithm. This is justified since these instructions take a constant time thus resulting in $k + O(N) = O(N)$. 
Robustness of Big-O

- The elementary set of instructions need to pre-defined to make O() robust.
- The Big-O notation is not conclusive of efficiency of actual programs, it can only be used for algorithms in general where N is very large.
- An algorithm of O(2N) is much worse than O(N) in real life.
- Also an algorithm of O(N^2) can still be faster than O(N) for some range of N.
- High-Performance Computing is an area in computing which tries to achieve speedup of existing algorithms.
- Note also that programs are made up of multiple algorithms resulting in the addition of the separate complexity of each algorithm. The end result is the largest O().
Space Complexity

- In addition to time complexity, each algorithm requires space in which to operate.
- We will not discuss this topic in depth since its quite complex.
- Sometimes time complexity can be traded-off with space complexity.
- Space complexity is notated also with the big O notation.

Reference:

\[ O(\log N) < O(N) < O(N \log N) < O(N^x) < O(N!) \]
Average Case Complexity

• Some algorithms have the same worst case complexity, yet might have a better average case complexity.
• The average case complexity is hard to find since one needs knowledge of the data distribution.
• The choice of algorithm is normally made on the average case complexity.
Upper and Lower Bounds

• The upper bound of an algorithmic problem is the complexity of the best algorithm that has been found.
• The lower bound of an algorithmic problem is the complexity of the best algorithm that will ever be found.
• The task of the computer scientist is to place the upper bound at the lower bound.
• Finding the lower bound without finding the algorithm is very hard.
• Harel gives an example of why the lower bound of a telephone book search is $O(\log_2 N)$ (each check needs to make a two-way comparison, thus setting the lower bound).
Terminology

- A **closed problem** is one whereby the lower bound algorithm has been discovered.
- An **algorithmic gap** is where the known algorithm is higher (at the upper bound) than the lower bound.
- Some problems have quite a huge algorithmic gap.