# Complexity

The previous chapters studied algorithms for problems and determined the cost of the algorithms. In this lecture, we study the problem and determine the minimum time required to solve the problem irrespective of a particular algorithm. The minimum cost to solve a problem is called the **complexity** of the problem.

If we know the complexity of a problem then we will know if any algorithm can be significantly improved. If the algorithm cost is the complexity of the problem then only minor improvements can be made.

There are a few techniques for finding the complexity of a problem.

## Trivial Lower Bounds

Frequently an inspection of the problem can suggest a lower bound for the complexity of the problem. If an algorithm runs at this lower bound then it is the complexity of the problem.

All algorithms must at least read the inputs to the problem and output the results. For example, a trivial lower bound for the complexity of evaluating a polynomial of degree $n$

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0$$

is $\Omega(n)$ because any algorithm must read the coefficient of arrays. We know that this must be the complexity of the problem meaning the lower bound is tight because Horner's rule evaluates the polynomial in $\Theta(n)$.

An example of a trivial lower bound based on the output of the problem is an algorithm generating all the permutations of an array must cost at least $\Omega(n!)$.

The difficulty with this technique is that it often suggests a lower bound that it is too low. For example, the trivial lower bound for the complexity of comparison based sorts is $\Omega(n)$, because the algorithm must read the array and output the results. This bound is too low.

Another difficult is that algorithms do not always have to read all the inputs to solve the problem. Consider the problem of searching for a key in an array. The augment that the algorithm must read the entire array suggests a complexity of $\Omega(n)$, but if the array is already sorted, then the algorithm does not have to read all of the array to find the key.

## Adversary Arguments

Adversary arguments for the lower bound can produce tighter lower bounds for the complexity of a problem. The adversary change the solution of the problem until it must give the correct answer. The solution produced by the adversary must be consistent with what the algorithm already knows or calculated.

Consider the problem of determining a number is between 1 and $n$ by asking questions, "is the number less than equal to $m$?" The adversary answers the questions so that finding the number will take as long as possible. This gives a lower bound of $\Omega(\lg n)$ for algorithm using a binary search approach.

Another example is to consider the cost to merge two sorted arrays, $a_i$ and $b_i$, both size $n$. Knuth explained that if the adversary uses the rule answer *true* to $a_i < b_i$ if and only if $i < j$. This forces the algorithm to produce the sequence

$$b_1 < a_1 < b_2 < a_2 < ... < b_n < a_n \quad (*)$$

This is consistent with $a_1 < a_2 < ... < a_n$ and $b_1 < b_2 < ... < b_n$ and the number of comparisons is $2n$-1. Note that each comparison is between elements of different arrays with equal or adjacent index values. If any

algorithm does not examine a comparison listed in (*) the adversary can switch the element of arrays. For example, if $a_i$ and $b_{i+1}$ was not examined then adversary can say the ordering is

$$b_1 < a_1 < b_2 < a_2 < ... < b_i < b_{i+1} < a_i < a_{i+1} < \quad ...< b_n < a_n$$

So a lower bound for merging sorted arrays $\Omega(n)$. We already have an algorithm that runs at $\Theta(n)$ so the complexity of merging sorted sorts $\Theta(n)$.

## Problem Reduction

The complexity of a problem can be determine by transforming the problem to another problem with know complexity.

The formula for the product of two numbers using only squares of numbers

$$x \bullet y = [(x+y)^2 - (x-y)^2]/4$$

illustrates that the product and squaring have the complexity.

## Infromation-Theoretic Arguments

Information-theoretic arguments examine the amount of information required to determine the answer.

Consider the problem finding a number between 1 and *n* by asking *yes* and *no* questions. Information theory explains that the amount of uncertainty is lg(*n*) where *n* is the number of possible choices. The answer from each question will generate one bit of information, lg(2) = 1. So lg(*n*) questions must be asked.

A related technique is to construct a **decision tree** to generate an algorithm

## *Decision Tree*

The technique using decision trees is two construct a decision tree that the algorithm uses for solving the problem. The leaves of the decision tree are all the possible solutions to the general problem. The algorithm begins at the root and travels down to the root of the tree to find the solution. Then the cost of the algorithm must be height of the tree, *h*

$$h \geq \text{ceiling}(\lg l)$$

where *l* is the number of leaves. If the decision tree is complete then the equality holds asymptomatically.

## Sorting

Illustrate the decision tree for sorting *a*, *b*, *c*.

Sorting an array with *n* elements using comparisions has *n*! possible solutions. So, the height of any decision tree must be at least lg *n*!. Using Stirling's formula

$$\lg n! \approx \lg \sqrt{[2\pi n]}\ (n/e)n = n \lg n - n \lg e + (\lg n)/2 + \lg 2\pi \approx n \lg n$$

So, at least $\Omega(n \lg n)$ comparison must be made. We already know algorithm with $\Theta(n \lg n)$ so the complexity of the problem is $\Theta(n \lg n)$.

## Searching an Ordered Array

Consider the problem of searching for a key in an *n* element array. The algorithm returns the index or bounding indices.
Illustrate the decision tree for a three element array

Note that each node has three children.

The number of possible solutions is $2n+1$ which include answers like $a[i] < key < a[i+1]$

The height of the tree is ceiling($\log_3(2n+1)$] $\varepsilon\ \Theta(\lg n)$