CS430

LECTURE NOTE NO.1

1 INTRODUCTION

Fundamental to solving a problem on a computer is the design of an *Algorithm* for solving the problem. In the context of computing an algorithm for solving a problem is a *finite sequence of well defined steps achieving the solution of the problem.*

A Simple Example:

For example, consider the problem of searching for a given number in a list of numbers, $a_1, a_2 \ldots a_n$. Assume that the numbers are given in a sorted order, i.e. $a_1 \leq a_2 \ldots a_n$.

To search for a given number in this list we use the following search function or algorithm. We let $SEARCH(x, L)$ be a function that returns true if x is in the list L and false otherwise. The following is a description of the search function

- Let $L = a_1 \ldots a_n$. Assume that $n > 0$ otherwise return with the answer false.
- If $x = a_m, m = n/2$ then we stop with the answer true.
- Otherwise **if** $x < a_m$ **then** $SEARCH(x, a_1 ... a_m)$ *else* $SEARCH(x, a_{m+1} .. a_n)$.

Note that in the above description we have used *recursion* to express the algorithm. We will consider ways of expressing algorithm using programmming languages later in the text. Essentially we have constructed a balanced binary tree, with all the numbers at the leaves of this tree.

An important issue in the design of algorithms is a proof of *correctness of the algorithm.*

We show that the algorithm $SEARCH(x, L)$ functions correctly : We need to prove the following claim:

Claim 1.1 *$SEARCH(x, L)$ correctly determines if $x \in L$ or if $x \notin L$.*

Proof:(outline) The proof is by induction on the number of elements in L.
Basis Step: When $n = 0$ the algorithm is trivially true.
Induction Step: Assuming that the algorithm works correctly when $n = k$, consider the case when there are $k + 1$ elements in the list. In this case the algorithm first compares with the median element. If $x = a_m$ then the algorithm returns the correct value immediately. Otherwise, if $x < a_m$ then the algorithm checks if the element x is in the left half of L else the element is checked to be in the right half of L. By induction these checks are correctly performed and the algorithm correctly detemines whether x is in the is L or not.

Another central issue that we want to tackle is the amount of time taken by this algorithm or the *time complexity* of the algorithm.

We shall estimate the number of operations required to search for x in L. The search requires comparisons and then repeated or recursive searches on a list half the size of the original list. To estimate the number of operations note that at each recursive step the size of the list is half the size of the original list. Thus in at most $log_2 n$ recursive steps the algorithm will have narrowed the search to a list of size 1. Thus the algorithm will require atmost $clog_2 n$, c a constant, comparisons to determine if $x \in L$.

An Interesting Search Game:

In many applications the cost of the search actually depends on the outcome. Consider the following interesting game:

Lopsided Search Game:

Suppose one has to search for a number in a sorted set. Everytime a guess is made, a penalty is incurred. However, the penalty of over-estimating the number is 2 units and under-estimating is only 1 unit. One can consider the following application: Imagine a factory floor where the scientist Dr. S Trength is interested in finding the breaking strength B of a metal bar. Of course one knows that the strength is at most U Kgs and, for simplicity, assume that it is integral. Applying a force over the breaking strength causes the bar to break, costing 2 units whereas applying a force less than B only costs only 1 unit. What would be the optimum strategy of discovering the breaking strength, i.e. what is the procedure of testing such that Dr. Trength incurs the least cost? There is an obvious solution where you discover the strength of the bar by an incremental test starting with a testing load of 1 Kgs and trying successive values of the force. What is the cost of this test? Is it optimum, the lowest cost that you can achieve?

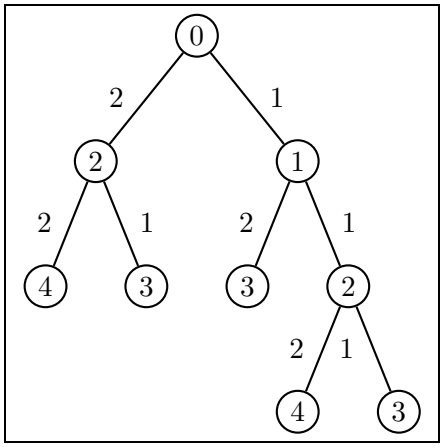
We can model the lopsided search problem as a binary tree with weights on the edges. The left branch is weighted one and the right branch has a weight of 2. The cost of a search path down the tree is the sum of the costs on the edges. The cost of a searching which terminates at a leaf l is the sum of the edge weights traversed in reacing the cleaf. We term this $Cost(l)$. The cost of the tree is specified as

$$\boxed{Cost(T) = \max_{l \in \mathcal{L}(T)} Cost(l)}$$

where $\mathcal{L}(T)$ is the set of all leaves of the tree T. How do we construct an optimal tree where we minimize the maximum length in the tree? This represents the worst case behavior of the algorithm.

We consider the structure of the optimal tree. Consider the weights at the leaves of an optimum tree. The first thing to notice is that the leaves cannot be weighted much differently. Suppose there is a leaf of weight $w-1$, called l_1 and another of weight $w+2$. The leaf of weight $w+2$ must have a sibling of weight $w+1$. Removing these two leaves and replacing l_1 with two leaves, which will have weight w and $w+1$, will result in tree where a leaf with weight $w+2$ has been eliminated. This would be a better tree and is preferred to the previous one. Repeating this for all

leaves of weight $w+2$ would result in a tree where either
(i) the weights of the leaves are either $w-1, w$ or $w+1$ or
(ii) the weights of the leaves are either $w, w+1$ or $w+2$.



An optimal Fibonacci tree for 5 outcomes

What is the cost of searching in a tree of with n leaves? We term this cost $C(n)$ To answer this question, consider M_w, the maximum number of nodes of weight w. Since a node of cost w is derived from nodes of cost $w-1$ and $w-2$ we get

$$M_w = M_{w-1} + M_{w-2}$$

where $M_0 = 1$ and $M_1 = 1$. This expression gives the familiar fibonacci sequence and we will see that $M_w = F_{w+1}$ where F_i is defined by:

$$\boxed{F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^i}$$

Now consider the first time that a node of weight w is generated. At that stage, the numer of leaf nodes in the tree, n_w is

$$n_w = M_{w-3} + M_{w-2}$$

since a node with weight $w-2$ is being first replaced and all leaf nodes with weight $w-1$ are generated from nodes with weight $w-3$. Thus the number of leaf nodes with weight $w-1$ equals the maximum (why?) number of nodes with weight $w-3$ and the number of nodes with weight $w-2$ equals $M(w-2)$ (Again why?). n_w is the fibonacci number F_w.

Thus the size of the tree, in terms of the total number of leaves, is also precisely a fibonacci number but only at certain stages, i.e. when a node of a certain weight is first present in the tree.

Thus when the number of leaf nodes, which corresponds to the possible strength values of the bar above, lies between F_{i+1} and F_i then the cost of determining the optimum strength of the bar is i.

$$\boxed{C(n) = i, \quad \forall n \text{ such that } F_i < n \leq F_{i+1}}$$

i.e. $C(n) = O(log_\Phi n)$

In general the two outcomes could cost α and β. The structure of the tree is much more complex in this case.

Time Complexity

Time complexity analysis of the algorithm is in fact a very important aspect in the study of algorithms since we are interested in *efficient* algorithms. In the above example a naive search which checks x with every element of the list L requires n comparisons.

In comparing two algorithms by their time complexity we will concentrate our attention on the asymptotic behaviour, i.e. the *asymptotic time complexity*. This is because comparisons are interesting primarily when the problem sizes (which is the size of the input list L in the above case) are large. Moreover we shall abstract out the constants in the time complexity since these constants are often unimportant as a first estimate of the complexity of the algorithm. The complexity of our algorithm for searching a list is thus expressed as $O(logn)$. We formalize this as follows: A function $g(n)$ is said to be $O(f(n))$ if there exists a constant c such that $g(n) \leq cf(n), \forall n \geq n_0$.

2 EXPRESSING ALGORTIHMS AS PROGRAMS.

The task of programming is a specification of the algorithm in a language interpretable by the computer.

We will use the following programming constructs to express our algorithms:

(1) The assignment statement:
variable ← expression

(2) The conditional statement:
if *condition* then *statement* else *statement*

(3) The iterative statements:
while *condition* do *statement*
repeat *statement* until *condition*

(4) A block of statements:

Begin
statement;
statement;
....
statement;
End;

(5) The procedural decalaration and invocation:
Procedure *name* (*parameters*) : *statement*

procedure-name(parameters)

(6) I-O statements: read *variable*; write *expression*;

3 MODELS OF COMPUTING

The algorithms that we devise will, expressed as programs, be implemented on computers. In the analysis of algorithms it is important to consider in detail the computer on which the program will be executed. For example, if we implement our algorithm on a parallel machine then it require less time to solve the problem. We are thus required to have a common platform to compare the performance of algorithms which solve a particular problem. Moreover, in the design of algorithms we would not like to take as a model any specific computing machine. Instead we will consider a class of machines. We will only consider sequential machines.

The first class that we can consider as a model of computing is our programming language itself. Assuming that each statement can be implemented in a unit of time the time complexity of our algorithm can be obtained by the number of statements that will be executed by the algorithm.

Unfortunately, this may be too powerful a model since a statement in the language can incude an arbitrary complex expression. A more commonly used model of computation is thus the RAM model.

3.1 RAM MODEL.

A RAM model comprises of an input device, an output device, a program (or computing device) and memory.

The computing device can accept a variey of simple basic instructions each of which is assumed to be computed in unit time. A typical list of these instructions reads as follows:

```
1. LOAD operand
2. STORE operand
3. ADD
4. SUBTRACT
5. MULT
6. DIVIDE
7. READ
8. WRITE
9. JUMP to location
10. HALT
```

This model of computation is called the *uniform cost RAM model*.

The model, unfortunately, appears too strong since additions and multiplications are assumed to be unit time operations and we are allowed to manipulate large numbers. In fact it is reasonable to measure the cost of operations by the cost of bit manipulations. Such a model is termed the *logarithmic cost RAM model*.

3.2 Decision Trees

A model that is especially useful when we are concerned with only the number of comparisions made by an algorithm is the decision tree model. We assume that comparisons have at most two outcomes.

A decision tree, $\mathcal{DT}$, is a binary tree such that each node corresponds to a comparison and each branch corresponds to an outcome of the comparison. Let $\mathcal{P}$ be a path from the root to a leaf node. At a leaf, L, is the outcome of the sequence of decisions $D(\mathcal{P})$ on the path $\mathcal{P}$.

The decision tree provides a useful way to measure the performance of algorithms. For any given input, the algorithm makes a sequence of comparisons. For each sequence of comparisons, S, leading to an outcome there is a corresponding path, $P(S)$, from the root to a leaf node in $\mathcal{DT}$. The cost of traversing this path is the number of comparisons made on the path, or equivalently the *length of the path*. (refer to the appendix for definitions and terminology)

The decision tree corresponding to the search procedure $SEARCH(x, L)$ is shown in Figure 1.

The cost of any search is at most $O(\log n)$.

4 Analyzing Algorithms

In this section we will discuss various measures of performance of algorithms. We need the following definitions dealing with asymptotic complexity.

Asymptotics

Let f and g be functions $\mathcal{N} \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set all natural numbers.

- f is $O(g)$ if
$$\exists c \in \mathcal{R}^+ \ s.t. \ f(n) \leq cg(n) \ \ \forall n \geq n_o$$
- f is $o(g)$ if
$$\lim_{n \rightarrow \inf} f(n)/g(n) \rightarrow 0$$
- f is $\omega(g)$ if
$$\lim_{n \rightarrow \inf} f(n)/g(n) \rightarrow \inf$$
- f is $\Omega(g)$ if
$$\exists c \in \mathcal{R} + \ \ s.t. \ f(n) \geq cg(n) \ \ \forall n \geq n_o$$
- f is $\Theta(g)$ if f is $O(g)$ and f is $\Omega(g)$.

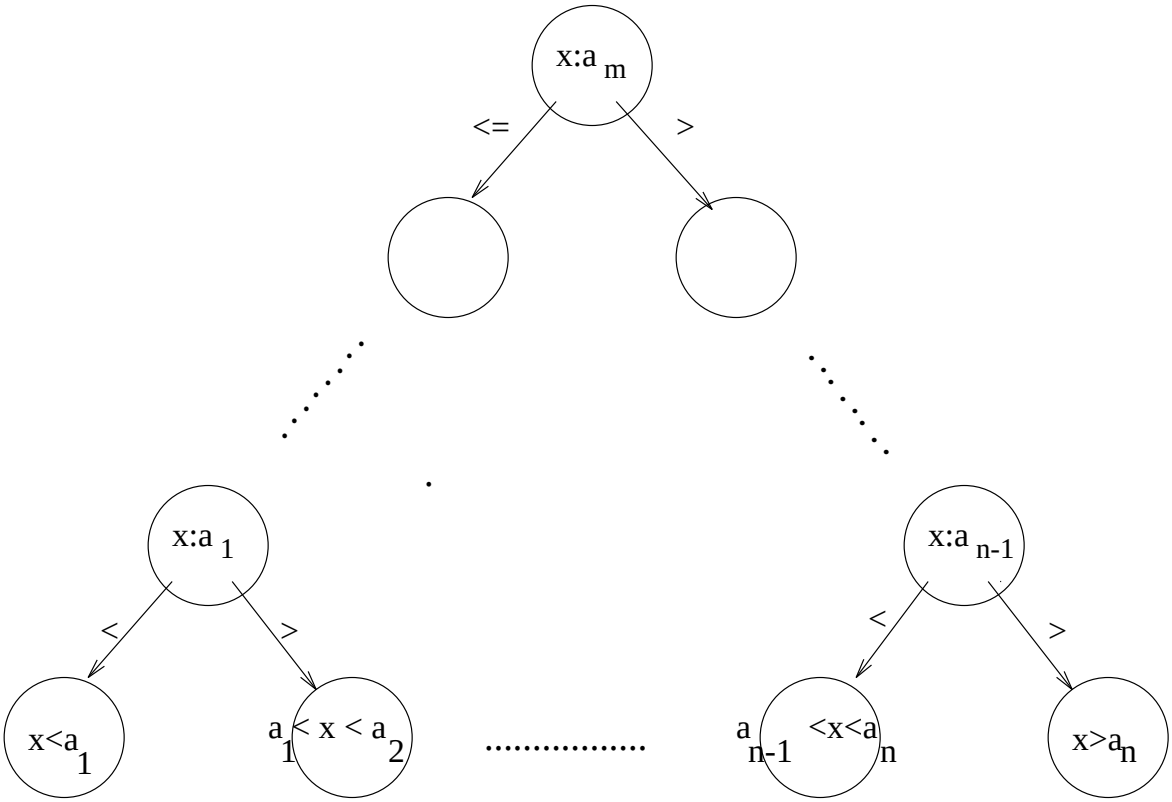


Figure 1: Decision Tree for the Search problem

Some of these asymptotics satisfy the properties of

- Transitivity:

  (i) If $f(n) = O(g(n)$ and $g(n) = O(h(n)$ then $f(n) = O(h(n)$.

  (ii) This property is also tue for $o(f)$, $\omega(f)$, $\Omega(f)$ and $\Theta(f)$.

- Reflexivity

  (i) $f(n) = \Theta(f(n)$

  (ii) $f(n) = O(f(n))$. The property is also true for Ω

- Symmetry

  (i) If $f(n) = \Theta(g(n)$ then $g(n) = \Theta(f(n))$

- Transpose Symmetry

  $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

  $f(n) = o((g(n))$ iff $g(n) = \omega(f(n))$.

Space and Time Bounds

Let $\mathcal{P}$ be a problem. And let $\mathcal{AP}$ be an algorithm that solves the problem $\mathcal{P}$. We let $\mathcal{IN}$ be the set of inputs that represent the problem. As an example, in the searching

problem, the parameters $(x, a_1, a_2 \ldots a_n)$ represent the input. Each input $I \in \mathcal{IN}$ has an associated size, represented by $|I|$. In this example that size would be n, the number of numbers in the search domain. An alternate mathod to measure size, which is often used in complexity theory, is simply to consider the number of bits in the input string. We let $\mathcal{AP}(I), I \in \mathcal{IN}$ be the outcome of the algorithm on the specific input I. More formally, the problem may be looked upon as a function mapping a domain $\mathcal{IN}$ to a range $\mathcal{OUT}$. In the search problem the answer is either *Yes* or *No*.

We will define two measures of the complexity of the algorithm. The *time complexity* and the *space complexity*. The *worst case time complexity* of any algorithm is defined as

$$T_A(n) = \max_{I \in \mathcal{IN}, |I|=n} [T_A(I)]$$

where $T_A(I)$ is the time required by the algorithm to solve the problem $\mathcal{P}$ on the input $I \in \mathcal{IN}$. Replacing $T(I)$ by $S(I)$ where $S(I)$ is the space required by the algorithm to solve the problem $\mathcal{P}$ gives the *worst case space complexity* of the algorithm.

EXAMPLE: In the search problem, the time complexity of $SEARCH(x, P)$ is $O(logn)$.

Suppose we have devised an algorithm to solve a problem. It would be of interest to determine the best algorithm for the problem. We formalize this below:

Let $\mathcal{AL}$ be the set of all possible algorithms that solve problem P. A *lower bound* on the time complexity of the problem may be defined as follows:

$$\mathcal{L}_{\mathcal{P}}(n) = min_{A \in \mathcal{AL}} T_A(n)$$

Similarly a lower bound on the space complexity of the problem can be defined as

$$\mathcal{L}_{\mathcal{P}}(n) = min_{A \in \mathcal{AL}} S_A(n)$$

Note that the lower bounds are parametrized with respect to the size of the input.

As an example we can show a lower bound on the complexity of the searching problem.

Claim 4.1 *The lower bound for the search problem is* $\Omega(logn)$.

Proof:left as an exercise.
Hint. Show that any binary decision tree that solves this problem must have height $\Omega(logn)$.

Consider the decision tree model which can be used to model search algorithms. What are the number of outcomes of the search problem? There are $n + 1$ possible outcomes for a failed search since the search parameter could fall into any one of the intervals defined by the n numbers in the list. Noting that the height of the decision tree corresponds to the worst case execution time of the algorithm, it thus suffices to find the mimimum possible height of a decision tree that models a search algorithm.

Note that any binary search tree of height k has at most 2^k leaf nodes. Thus every search tree with $n + 1$ leaf nodes must have height $\Omega(\log(n + 1))$. Hence the lower bound.