Lower Bound Techniques

We need to learn some techniques to establish that a given algorithm is the most efficient possible. The way this is done is by discovering a function, *g(n),* which is a lower bound on the time that *any* algorithm must take to solve the given problem. If we have an algorithm whose computing time is the same order as *g(n)* then we know that asymptotically we can do no better.

Deriving good lower bounds is often more difficult than devising efficient algorithms. Perhaps this is because a lower bound states a fact about all possible algorithms for solving a problem. Usually we cannot enumerate and analyze all of these algorithms, so lower bound proofs are often hard to obtain.

However, for many problems it is possible to easily observe that a lower bound identical to *n* exists, where *n* is the number of inputs (or possibly outputs) to the problem. For example consider all algorithms which find the maximum of an unordered set of *n* integers. Clearly every integer must be examined at least once and so $\Omega(n)$ is a lower bound for any algorithm which solves this problem. Or, suppose we wish to find an algorithm which efficiently multiplies two n x *n* matrices. Then $\Omega(n^2)$ is a lower bound on any such algorithm since there are $2n^2$ inputs which must be examined and n^2 outputs to be computed. Bounds such as these are often referred to as *trivial* lower bounds because they are so easy to obtain. We know how to find the maximum of *n* elements by an algorithm which uses only *n*- 1 comparisons so there is no gap between the upper and lower bound for this problem.

Comparison trees to estimate lower bounds

Comparison trees are useful for determining lower bounds for sorting and searching problems. We will see how these trees are especially useful for modeling the way in which a large number of sorting and searching algorithms work. By appealing to some elementary facts about trees the lower bounds are obtained. Suppose that we are given a set *S* of distinct values upon which an ordering relation " < " holds.

The *sorting problem* calls for determining a permutation of the integers 1 to *n,* say p(1) to *p(n)* such that the *n* distinct values from *S* stored in A(1:n) satisfy A(p(1)) < *A(p(2))* < ... < *A(p(n))*.

The *ordered* ***searching*** *problem* asks if a given element x ∈ *S* occurs within the elements in A(1:n) which are ordered so that *A(1)* < ... < *A(n)*. If x is in *A(1:n)* then we are to determine an *i* between 1 and *n* such that *A(i)==x*.

The ***merging*** *problem* assumes that two ordered sets of distinct inputs from *S* are given in A(1:m) and B(1:n) such that *A(1)* < ... < *A(m)* and *B(1)* < ... < *B(n);* these *m* + *n* values are to be rearranged into an array C(1: m + *n)* so that *C(1)* < ... < *C(m + n)*.

For all of these problems we will restrict the class of algorithms we are considering to those which work solely by making comparisons between elements. No arithmetic involving elements is permitted, though it is possible for the algorithm to move elements around.

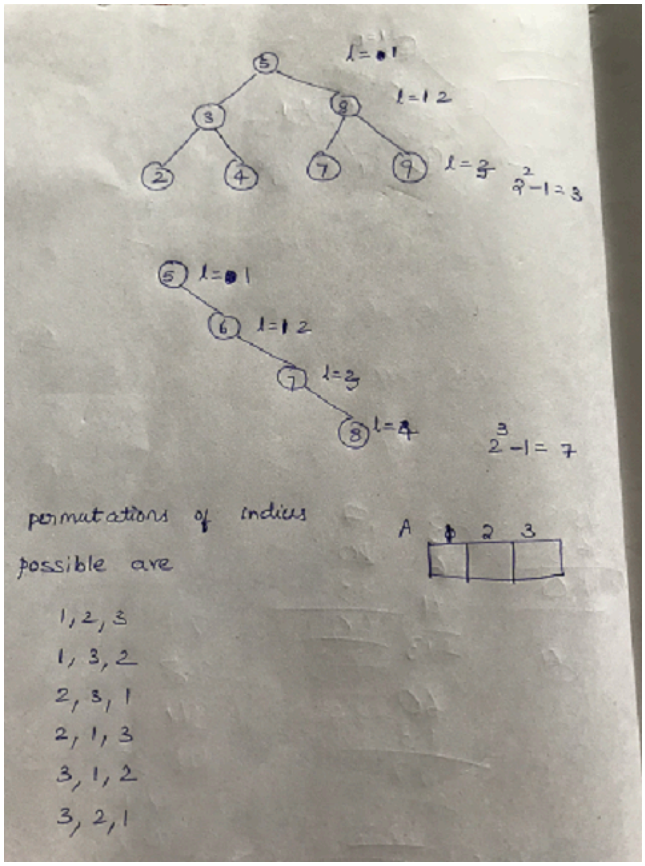
This class of algorithms is referred to as *comparison based* algorithms. We rule out algorithms such as radix sort which decompose the values into subparts.

In obtaining the lower bound for the ordered searching problem, we shall consider only those comparison based algorithms in which every comparison between two elements of A is of the type "compare x and *A{i)".* Any searching algorithm which satisfies this restriction can be described by an extended binary tree. Each internal node in this tree represents a comparison between x and an *A(i).* There are three possible outcomes of this comparison: x < *A(i),* x = *A(i),* and x > *A(i).* We may assume that if x = *A(i)* then the algorithm terminates. Hence the progress of the algorithm may be described by a binary tree in which the left branch is taken if x < A*(i)* and the right branch is taken if *x > A(i).* If the algorithm terminates following a left or right branch (but before another comparison between x and *A(i))* then no *i* has been found such that x= *A(i)* and the algorithm must declare the search unsuccessful.

Figure 1 shows two comparison trees, one modeling a linear search algorithm and the other a binary search. It should be easy to see that the comparison tree for any search algorithm must contain at least n internal nodes corresponding to the n different values of *i* for which x = *A(i)* and at least one external node corresponding to an unsuccessful search.

Theorem : Let ***A(1:n),*** n > 1, contain n distinct elements, ordered so that *A(1)* < ... < *A(n).* Let FIND(n) be the minimum number of comparisons needed, in the worst case, by any comparison based algorithm to recognize if x ∈ A(1 *:n).* Then FIND(n) >=ceil(log (n + 1)) .

Proof: Consider all possible comparison trees which model algorithms to solve the searching problem. FIND(n) is bounded below by the distance of the longest path from the root to a leaf in such a **tree.** There must be n internal nodes in all of these trees corresponding to the n possible successful occurrences of x in *A.* If all internal nodes of a binary tree are at levels less than or equal to *k,* then there are at most 2^k — 1 internal nodes. Thus n <= 2^k- 1 and FIND(n) >=ceil(log (n + 1) .)



From the above theorem we can conclude that binary search is an optimal worst case algorithm for solving the searching problem.

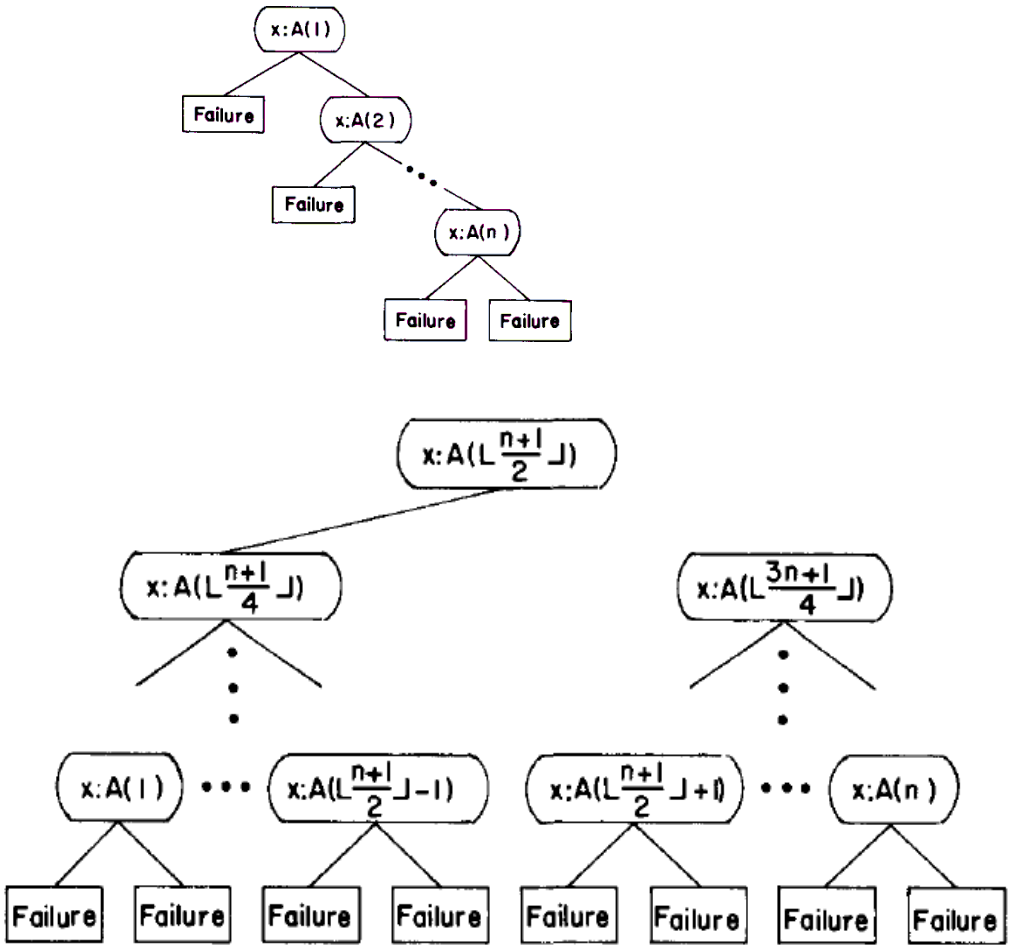


Fig 1 Comparison trees for two searching algorithms

Sorting Problem:

Since the keys are distinct, any comparison between *A(i)* and A(j) must result in one of two possibilities: either *A(i) < A(j)* or *A(i) >A(j).* Thus this tree will be a binary tree where the value of any internal node is the pair *i:j* which represents the comparison of *A(i)* with *A(j).* If *A(i)* is less than *A(j)* then the algorithm proceeds down the left branch of the tree and otherwise it proceeds down the right branch. The external nodes represent termination of the algorithm. Associated with every path from the root to an external node is a unique permutation. To see that this permutation is unique, note that the algorithms we allow are only permitted to move data and make comparisons. The data movement on any path from the root to an external node is the same no matter what the initial input values are. As there are n! different possible permutations of n items, and any one of these might legitimately be the

only correct answer for the sorting problem on a given instance, the comparison tree must have at least n! external nodes.

Figure 2 shows a comparison tree for sorting 3 items. The first comparison is *A(1)* : A(2). If *A(1)* is less than A(2) then the next comparison is A(2) with A(3). If A(2) is less than A(3) then the left branch leads to an external node containing 1,2,3. This implies that the original set was already sorted for *A(1)* < A(2) < A(3). The other five external nodes correspond to the other possible orderings which could yield a sorted set.

We consider the worst case for all comparison based sorting algorithms. Let *T(n)* be the minimum number of comparisons which are sufficient to sort n items in the worst case. Using our knowledge of binary trees once again, if all internal nodes are at levels less than *k* then there are at most 2^k external nodes, (one more than the number of internal nodes). Therefore, letting *k = T(n)*

$$n! \leq 2^{T(n)}$$

Since *T(n)* is an integer we get the lower bound

$$T(n) > \lceil \log n! \rceil$$

By Stirling's approximation it follows that

$$\log n! = n \log n - n / \ln 2 + (1/2) \log n + O(1)$$

where ln 2 refers to the natural logarithm of 2 while log n is the logarithm to the base 2 of n. This formula shows that *T(n)* is of the order n log n.

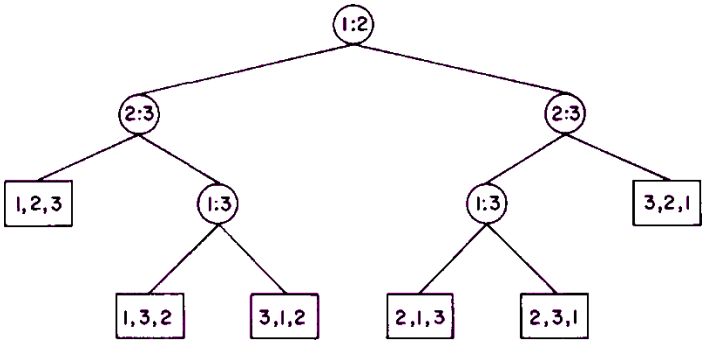


Figure2 : A comparison tree for sorting three items

Hence we say that no comparison based sorting algorithm can work in fewer *than* O(n log n) *time*

Binary Insertion sort and Merge Insertion sort (Ford-Johnson algorithm) come very close to the lower bound for small values of n.

In binary insertion sort we find the right place for insertion of a new element by applying binary search on already sorted data.

Implementation of Binary Insertion Sort

```
#include<stdio.h>
void binaryinsert(int a[],int n,int i);
main()
{
int a[10]={30,10,100,40,20,80,60,90,50,70},i,n=10;

for(i=1;i<=n-1;i++)
binaryinsert(a,n,i);
for(i=0;i<=n-1;i++)
    printf("%d ",a[i]);
}
void binaryinsert(int a[],int n,int i)
{
    int low,high,mid,j,key;
    key=a[i];
    low=0;high=i-1;
    while(low<=high)
    {
        mid=(low+high)/2;
        if(key<a[mid])high=mid-1;
        else
            low=mid+1;
    }
    for(j=i-1;j>=low;j--)
    a[j+1]=a[j];
    a[low]=key;
}
```