**CS330 – Search Analysis and Lower Bounds**

**Purpose:** Now that we know how to analyze the running times of algorithms it is time to start doing just that! We will start with the basic scenario of searching for a value in array. Once we have an algorithm we also want to know if this is the optimal, or most efficient way to accomplish the task. Proving a "lower bound" shows that any algorithm for the task, even algorithms we haven't thought of yet, must do at least that much work.

**Knowledge:** This activity will help you become familiar with the following content knowledge:

- Analyzing search algorithms
- Proving lower bounds for search

**Activity:** With your group perform the following tasks and answer the questions. You will be reporting your answers back to the class in 60 minutes.

1. Consider our old friend, the linear search:

```
int search(int[] list, int target, int n)
 {
   for (int i=1; i<=n; i++)
       if (target == list[i]) return i;
   return -1;
 }
```

   What is the order of growth for the worst case?

2. If our list is unordered you probably have a gut feeling that this is the best we can do for our search. It would be nice, however, to actually know this for certain so I am going to attempt a proof that we can't do any better than $T(n) = n$ by showing that any algorithm with less than $n$ comparisons will be incorrect for some list $L$:

   Suppose a search algorithm has $n - 1$ comparisons (or less) of the target to the list. Then there exists and element of $L$ which is never compared to the target. So if that is the one that we are searching for, the algorithm fails.

   Can you find a flaw in that argument? There is one!

   Hint: You could compare elements in the list with each other. Why could this preprocessing reduce the amount of comparison with the target? What if $L$ is sorted?

3. A proof that ANY algorithm to solve a problem must do a certain amount of work is called a lower bound proof. Showing a lower bound can be challenging because it needs to consider all possible algorithms, even ones we might not have thought about.

   Here is a second try for a lower bound for searching an unordered list $L$:
   Suppose that an algorithm does $k$ comparisons for preprocessing on $L$ and $j$ comparisons with the target. The preprocessing splits $L$ into $m$ "pieces". A piece is simply a collection of elements which are "connected" by comparisons. In other words if you drew a connecting line between elements which have been compared to each other, a piece would be connected with these lines.

   What is the minimum number of comparisons $k$ that must be performed to have $m$ pieces?
   Given that we have $m$ pieces, what is the minimum number of comparisons $j$ needed with the target?
   Why does this show that any algorithm needs at least $n$ comparisons in total?

4. We will now consider searching in a sorted list, rather than an unordered list. It is reasonable to expect that we will be able to do the search more efficiently, given that the list is sorted. Here is an algorithm to do that called Binary Search. Make sure you understand what this algorithm is doing.

```
int Bsearch(int[] list, int target, int low, int high){
   if (low == high)
       if  target == list[low]) return low;
       else return -1;
   else {
     mid = floor((low+high)/2);
     if (target > list[mid]
          return Bsearch(list,target,mid+1,high);
    else
          return Bsearch(list,target,low,mid);
    }
 }
```

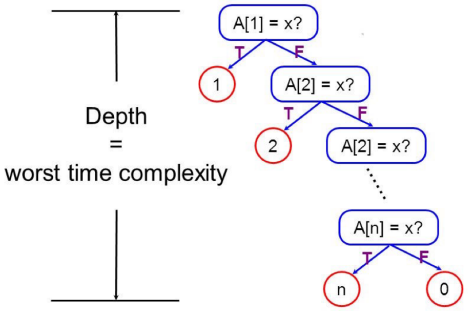
   Give the recurrence relation for the worst case number of comparison tests with the target. (You may assume that we are dealing with powers of 2 so that the division by 2 always results in an integer).

5. We can determine the worst case for binary search by solving the recurrence relation:

   $T(n) = T(n/2) + 1, T(1) = 1$

   Get the closed form to do that.

6. One way to look at searches is with a "decision tree". Here is a decision tree for linear search.




   What would a decision tree for Binary Search look like?

7. Given ANY algorithm for search, we can look at the decision tree to determine the worst case. First we need to determine some facts about trees.

   Use induction to show that a binary tree of height $h$ has at most $2^{h+1} - 1$ nodes. Hint: Induct on the height of the tree.

8. What is the minimum height of a decision tree with $n$ nodes? Why? Does this prove that binary search is optimal?

9. But wait.. the optimality of binary search is assuming that all we can do is compare values. What if...

   (a) the data is not sortable. (Can you give such a scenario?)

   (b) the data is sorted but in a data structure where it does not cost the same to get at each location (What data structure would have that property?)

   (c) the data is static so we know all possible search requests (Do I see a hash table?!)