Q1)

Given an array A which stores 0 and 1, such that each entry containing 0 appears before all those entries containing 1. In other words, it is like {0, 0, 0, ..., 0, 0, 1, 1, ..., 111}. Design an algorithm to find out the small index i in the array A such that A[i] = 1 using c log n instructions in the worst case for some positive constant c.

S1) It's another form of peak finder problem that we discussed in class, We exploit the idea used in binary search.

Here is the function to implement the binary search method.

```
void First_index_with_one(int[] A, int n)
{
    if (A[0] == 1)
        printf("The first index storing ONE is 0");

    else if (A[n - 1] == 0)
        printf("All entries of A are 0");

    else //There is a unique i such that A[i-1]==0 and A[i]==1
    {
        int left = 0;
        int right = n - 1;
        int Isfound = 0;

        while (Isfound == 0)
        {
            mid = (left + right) / 2;
            if (A[mid] == 0)
                left = mid + 1;
            else
            {
                if (A[mid - 1] == 0)
                    is_found = 1;
                else
                    right = mid - 1;
            }
        }

        printf("The first index containing one is %d", mid);
    }
}
```

Q2)Theorem All horses are the same color

Proof by induction

Base Case: n = 1, then there is only one horse in the set, and it must consequently have the same color as itself.

Inductive Step: Let us assume for n ≥ 1 that all horses in a set of size n have the same color.

FInal step: suppose we have a set of n + 1 horses: { h_1, h_2, ..., h_n, h_{n+1}}. We could break this up into to two sets, { h_1, h_2, ..., h_n} and { h_2, ..., h_n, h_{n+1}}. Each of these is a set of size n and thus all horses in each, by assumption, are the same color. As there is an overlap between the two sets, it follows that all the horses in both sets must be the same color, and thus, all the horses in {h_1, h_2, ..., h_n, h_{n+1}} have the same color.

Choose the correct option:

A) I always knew that Math is unreliable

B) Induction is useful for numbers

C) Induction doesn't always work

D) All the other options are wrong. The solution given is wrong because

_____

S2)

FInal step: suppose we have a set of n + 1 horses: { h_1, h_2, ..., h_n, h_{n+1}}. We could break this up into to two sets, { h_1, h_2, ..., h_n} and { h_2, ..., h_n, h_{n+1}}. Each of these is a set of size n and thus all horses in each, by assumption, are the same color. As there is an overlap between the two sets, it follows that all the horses in both sets must be the same color, and thus, all the horses in {h_1, h_2, ..., h_n, h_{n+1}} have the same color.

The underlined part in the final step should is indicative of the fact that there is something wrong in this solution: intuitively you should realize the reasoning given (the highlighted part) is not correct. For example, what if { h_1, h_2, ..., h_n} and h_{n+1} is be a black horse?

Base Case: n = 1, then there is only one horse in the set, and it must consequently have the same color as itself.

Inductive Step: Let us assume for n ≥ 1 that all horses in a set of size n have the same color

This inductive step is not justified for this base case. Therefore, the inductive step is wrong. Therefore, the final step is wrong

The proof is wrong because the base case is wrong.

You can prove amazing things if the base case is wrong.

Q3)

Here are three sorting algorithm listed below with basic explanation and Pseudocode, read and answer following questions.

Insertion

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Pseudocode:

```
for i = 1 to length(A) - 1
     j = i
      while j > 0 and A[j-1] > A[j]
      swap A[j] and A[j-1]
      j = j - 1
      end while
end for
```

BogoSort

If bogosort were used to sort a deck of cards, it would consist of checking if the deck were in order, and if it were not, throwing the deck into the air, picking the cards up at random, and repeating the process until the deck is sorted

Pseudocode:

```
while not isInOrder(deck):    shuffle(deck)
```

Note: deck is your unordered array containing n elements.

Merge Sort

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into *n* sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

Pseudocode: a is an array containing n elements.

```
func mergesort( var a as array )
     if ( n == 1 ) return a
     var l1 as array = a[0] ... a[n/2]
     var l2 as array = a[n/2+1] ... a[n]
     l1 = mergesort( l1 )
     l2 = mergesort( l2 )
     return merge( l1, l2 )
end func

func merge( var a as array, var b as array )
     var c as array

     while ( a and b have elements )
          if ( a[0] > b[0] )
               add b[0] to the end of c
               remove b[0] from b
          else
               add a[0] to the end of c
               remove a[0] from a
     while ( a has elements )
          add a[0] to the end of c
          remove a[0] from a
     while ( b has elements )
          add b[0] to the end of c
          remove b[0] from b
     return c
end func
```

Question:

What are the time complexities of these algorithms in worst case in the form of big omega? For bogosort, find the average case.

For n=2, n=5 and n=100, rate these three algorithms from fastest to slowest

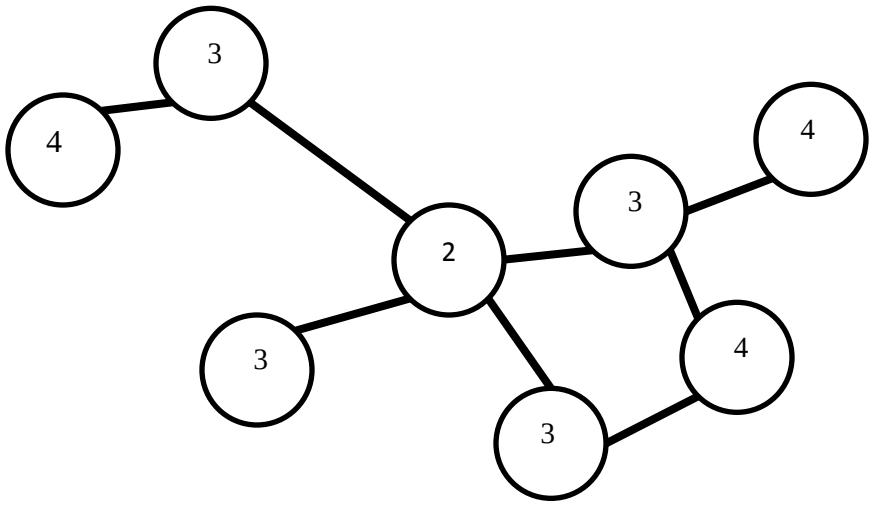
S3) Answer: insertion: O(n^2), bogosort: O((n+1)!), mergesort: O(n*log(n))

n=2 merge<bogosort<insertion

n=5 merge<insertion<bogosort

n=100 merge<insertion<bogosort

Q4) Graphs and Centrality



Centrality is an indicator of how connected a node in a graph is. One definition of a node's centrality is defined as the largest number of edges that need to be crossed from the node to reach any other node on the graph using the shortest path to the node. A lower value means the node is more central.

Create the procedure (or pseudocode) for calculating the centrality of a node from a graph.

S4)

1. Create a set of visited and adjacent (or not visited) nodes, and an integer value x.
2. Assign all nodes a temporary value (INT_MAX or similar large).
3. Put the start node in the set of adjacent nodes with a new value of 0.
4. Loop while there are nodes in the adjacent set:
    a. Pick a node from the adjacent set.
    b. Assign a variable n to be the value of the node.
    c. If n is greater than x, set x to be n.
    d. Remove node from adjacent set and add to visited set.
    e. For each node adjacent to this node:
        i. If the node is in the visited set with a value less than n+1, go to next node.
        ii. If the node is in the visited set with a value greater than n+1:
            1. Remove the node from the visited set.
        iii. Add the node to the adjacent set with a value of n+1.
5. The start node's centrality value is x.

Q5) a binary tree is a data structure that facilitates searching and inserting into a data set. A common problem that programmers encounter is to poorly balance a binary tree as shown in Fig 5.1. To fully understand the importance of balancing a binary tree, answer the following questions:

a) What is the worst case performance of finding an element in a balanced binary tree and of an extremely unbalanced binary tree* ? (in big-o-notation)

b) What data structure does an extremely unbalanced binary tree* resemble ?

*extremely unbalanced tree means that all data points are always inserted into either the left or right of the data point before

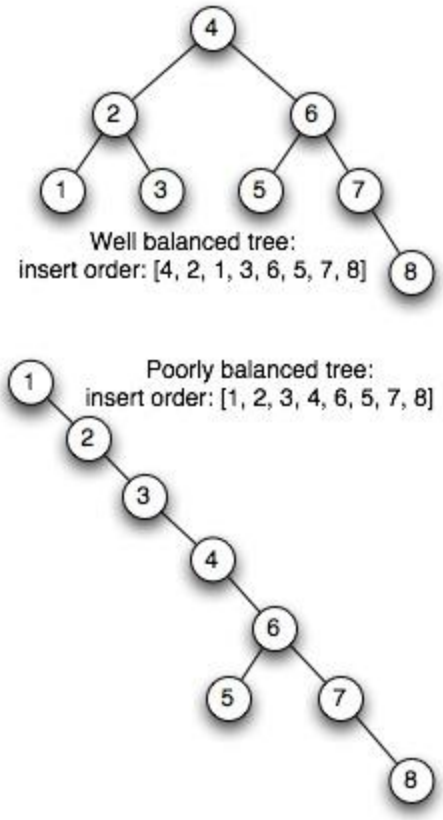


Fig 5.1: balanced vs unbalanced binary trees

S5)

a) O(log n) vs O(n)

b) singly linked list