1. In our discussion of program efficiency, we listed two resources that were of interest to us: (a) space, the amount of storage required by an application; and (b) time, the execution time required for an application to solve a particular problem. Although our focus has been on time complexity of various data structure schemes (as suggested in your notes), we had mentioned a recurring theme throughout the whole of computer science where space complexity comes back into the picture anyway. We refer to the time-space tradeoff. Explain this notion using concrete examples if appropriate. (10 pts.)

"Time-space tradeoff" describes the virtual tautology that what you gain in time resource, you give back in terms of space requirements, and vice-versa. That is, when developing a software solution, a more efficient algorithm (time-wise) usually requires more working memory (space requirements). A good example of this would be the array-based and node-based implementation of lists. We trade the efficient index-based performance of arrays ($O(1)$) with the fact that we allocate array space even though we may not have list elements to store in that space. What we gain in space economy with node-based implementations, we give back in the fact that index-based list access is invariably an $O(n)$ proposition.

There are many other examples and situations where this tradeoff seems to be the rule rather than the exception. That being said, it is not true however that if one had an unlimited amount of space, one can then reduce the time complexity of a problem solution without bound. Many problems have intrinsic difficulty and no amount of additional space may reduce the amount of time needed to solve it.
2. What is the difference between the notion of complexity of a method and complexity of a problem? How do these notions come together when we determine whether a method is optimal or not? Use “big O” notation in your discussion (10 pts.)

Method complexity is a measure of the resource required by the method in carrying out its intended action. Usually it is time resource that we measure. So method complexity in this context will be a measure of the amount of time that the method will need in order to perform its intended action. We express this using the big-O notation, e.g., $\mathcal{O}(n)$ for the remove() method for array-based implementation of lists.

Problem complexity, on the other hand, is a measure of the intrinsic difficulty of a problem and expresses the amount of resource (again, usually it is time that we are concerned about) required in solving the problem because of its innate complexity. We express this using the "omega" notation, e.g., $\Omega(n \log n)$ for the sorting problem. This means that no matter what method logic is used, the sorting method will require time resource that grows as fast as the function $n \log n$ where $n$ is the number of values to be sorted.

A method is deemed "optimal" when its method complexity matches the complexity of the problem that it seeks to solve. Examples of this would be merge sort and heap sort, two sorting methods which have complexities that match the problem complexity of sorting.
3. Define what a binary search tree (BST) is. (5 pts.)

A binary search tree (BST) can be defined recursively:
- an empty binary tree is a BST;
- a binary tree whose root element is greater than all of the elements in its left subtree but less than all of the elements in its right subtree, both of which are BSTs themselves, is itself a BST.

4. Is the following tree a BST? Why or why not? (5 pts.)

This is not a binary search tree. The right subtree contains the value 1, which is not greater than the root value of 3.
5. Can one define a ternary search tree, i.e., one where the nodes can have three children? Why or why not? (5 pts.)

A ternary search tree would naturally store in the middle subtree all the values that are equal to the root element. This, of course, is not allowed in a dictionary, where each key is unique. Thus, the middle subtree would be empty and would be unused. However, there may be cases where the "key" might be repeated (we call them homonyms) so the middle subtree might in fact be useful. In practice, homonyms are the exception rather than the rule, so if we ever had ternary search trees, the middle substructures would be "shriveled," much smaller than the left and right substructures:

6. BSTs evolve according to the order in which the items are inserted in it. For example, the sequence:

three, blind, mice, see, how, they, run

would result in the following BST:

```
three
  
blind
  
mice
    
how
  
see
    
run
    
they
```
(cont.)

Draw the BST that would result if the words were inserted in reverse order, i.e.,

run, they, how, see, mice, blind, three

Use the space below for your answer. (10 pts.)

7. Supply another ordering of the same seven words that would result in the same BST when inserted in the order in which they came. (5 pts.)

run, how, they, blind, mice, see, three
8. Discuss what *hash functions* are and indicate their usefulness in implementing dictionaries. (5 pts.)

*Hash functions* assign to keys an address that can then be used to index a table where the key-value pair could be stored. We illustrate this below:

```
key ─── Hash function ─── address
```

This can then be used to implement a dictionary, retrieving the key-value pair by computing the hash function for the key and then using the value to "peek" into the table with that value as index. Ideally, the key-value pair would be located at that address. However, there can be key values that are assigned by the hash function the same address, what is called a "collision." In this case, there are many strategies employed to resolve the issue. These include re-hashing and various kinds of probes (linear, non-linear). At the expense of unused table locations, the performance of hash-based dictionaries would be close to $O(1)$ for retrieves, adds, and deletes.

9. Suppose that the keys to be used in our hash table are integers within the range 100,000 and 999,999 inclusive. The hash function we choose “folds” the keys by taking the two high order digits (the hundred-thousands’ digit and the ten-thousands’ digit) and then affixing the two low order digits (the tens and units digits) to obtain a value within the range 1,000 and 9,999. For example, the key value 235,976 will be hashed into the value 2,376 while the key value 123,456 will be hashed into the value 1,256. You are to supply the implementation of this hash function. We supply the specification below: (10 pts.)

```c
/**
 * Hash value for the specified key.
 */
int hash ( int key ) {

    int answer = key / 10000;  //shift digits 4 to right
    answer = answer * 100;      //shift digits 2 to right
    answer = answer + key%100;  //add last 2 digits of key
    return answer;
}
```
10. The text describes a simple array-based implementation of binary trees (cf. Sec. 23.5.1). In the description, we are told that the root of the binary tree is stored in array element with index 0 and if a node is stored in array element \( n \), then its left child is stored in array element \( 2n+1 \) while its right child is stored in \( 2n+2 \). For example, the following array configuration may represent the binary tree illustrated further below:

Using this as the basis of our implementation, we can supply the specs of a “bounded” binary tree:

```java
//Constructor(s)
/**
 * Construct an empty binary tree with default size.
 */
public BoundedBinaryTree() { ... }

/**
 * Construct an empty binary tree whose maximum capacity is equal to the
 * supplied parameter.
 */
public BoundedBinaryTree(int size) { ... }
...

//Queries
/**
 * Is the BoundedBinaryTree empty?
 */
public boolean isEmpty() { ... }

/**
 * The item at the root of the BoundedBinaryTree.
 */
public Object root() { ... }
```
/**
 * The left subtree of the BoundedBinaryTree.
 */
public BoundedBinaryTree left() { ... }

/**
 * The right subtree of the BoundedBinaryTree.
 */
public BoundedBinaryTree right() { ... }

//private components
private Object[] theArray ;  //holds the items stored in the BoundedBinaryTree
private int size ;           //the number of items in the BoundedBinaryTree
}

The actual implementation of this class runs into real practical problems. Notwithstanding, for purposes of mental exercise, let us develop some of the methods in this class. The constructors are especially instructive:

/**
 * Construct an empty binary tree with default size.
 */
public BoundedBinaryTree () {
    theArray = new Object[ DEFAULT_SIZE ] ;
}

/**
 * Construct an empty binary tree whose maximum capacity is equal to the
 * supplied parameter.
 */
public BoundedBinaryTree ( int size ) {
    theArray = new Object[ size ] ;
}

Supply the implementation of the method isEmpty() as well as root(). We provide the specs and the space for the rest of the implementation below: (10 pts. each)

/**
 * This is the empty BoundedBinaryTree.
 */
public boolean isEmpty () {

    return this.size == 0;

}

//more
/**  
 * The item at the root of the BoundedBinaryTree.  
 */

public Object root () {

    assert (!this.isEmpty() );
    return this.theArray[ 0 ];
}

}

How would you implement the method height() as defined previously in class and during the last test. Give your answer below: (15 pts.)

/**  
 * The height of the BoundedBinaryTree  
 */

public int height () {

    if ( this.isEmpty() )
        return 0;
    else {
        int leftHeight = this.left().height();
        int rightHeight = this.right().height();
        return Math.max( leftHeight, rightHeight ) + 1;
    }
}

} //end height