1. A typical harddisk has an average access time of about 5 milliseconds \((5 \times 10^{-3} \text{ seconds})\), and a typical main (RAM) memory access time is 50 nanoseconds \((50 \times 10^{-9} \text{ seconds})\).

a) How many times faster is a main memory access than an average harddisk access?

b) What advantage(s) does a harddisk have over main memory?

2. If we are sorting a collection of \(n\) data items using a simple sort (e.g., selection or bubble sort), we need about \(n^2/2\) data accesses. An algorithm for the main program might be:
   i) Read all \(n\) data items into an array (in memory) from disk
   ii) Sort the data items in the array
   iii) Store the sorted data items back to disk

a) In the above algorithm, why are the \(n\) data items on the disk to start with?

b) In the above algorithm, why don’t we just sort the data items on the disk and avoid moving them to an array in memory?

3. Last semester in Introduction to Computing we covered chapters 1-12 which mostly focused on C features, except:
   - pass-by-reference parameter passing for function arguments
     
     **Pass-by-reference Example:**
     ```cpp
     void getValue(int & value) {
         cout << "Enter a value: ";
         cin >> value;
     } // end getValue
     
     Calling of function:
     getValue(myIntData);
     ```

     **C Pass-by-value using Pointers Example:**
     ```cpp
     void getValue(int * value) {
         cout << "Enter a value: ";
         cin >> *value;
     } // end getValue
     
     Calling of function:
     getValue(&myIntData);
     ```
   - used some C++ objects from libraries: I/O - cin, cout, ifstream, ofstream, and C++ strings e.g., cin.getline(cString, SIZE)

This semester you’ll learn how to create your own classes/objects for (in-memory) data structures (e.g., stacks, queues, binary-search trees, graphs) and algorithms that use these data structures. (NOTE: We
save object-oriented design (OOD) for Intermediate Computing (810:053) with this course being more procedural or object-based design.

a) If you download the receipe for a cake from the Internet, how many cakes do you have?

b) How many cakes can you make from this receipe?

c) In object-oriented languages like C++, programmers define classes (“the receipes”) and can make (construct) as many objects (class instances) as needed. Consider the following main program that uses a simple Die class. How many Die objects are used in this program?

```cpp
/* File: TestDieMain.cpp to test the Die class */
#include "Die.h"
#include <iostream>
using namespace std;

int main() {
    Die die1 = Die(); // 6-sided die
    Die die2 = Die(8); // 8-sided die
    cout << "die1: " << die1.getRoll() << " die2: "
         << die2 << endl;
    ++die1;
    cout << "++die1 " << die1 << endl;
    cout << "Rolls two dice 10 times: " << endl;
    for (int count=0; count < 10; count++) {
        die1.roll();
        die2.roll();
        cout << die1 << " " << die2;
        if (die1 == die2) {
            cout << " dice are equal";
        } // end if
        cout << endl;
    } // end for
} // end main
```

die1: 5 die2: 7
++die1 6
Rolls two dice 10 times:
  3 5
  5 1
  3 1
  2 5
  1 1 dice are equal
  3 7
  5 5 dice are equal
  1 2
  1 1 dice are equal
  3 4
/* Declaration of Die class */

#ifndef DIE_H
#define DIE_H
#include <iostream>
#include <iomanip>
using namespace std;

class Die {
    // These needed to be implemented as non-member functions since the left operand is not a Die object. We make them friend functions to allow access to the private data members.
    friend ostream & operator<<( ostream &, const Die &);  
    friend istream & operator>>( istream &, Die &);

private:
    int numberOfSides;
    int currentRoll;

public:
    Die(int sides = 6);  // default constructor with default sides of 6
    void roll();    // rolls the die
    int getRoll() const;  // returns the value of the current roll
    bool operator==( const Die &) const;
    Die & operator++();
};  // end Die class
/********************************************************** Die.cpp **********************************************************/

* Implementation of die class
**********************************************************/

#include
#include <cstdlib>
#include <ctime>
#include <cassert>
#include <iostream>
using namespace std;

// constructs a die with the specified number of sides
Die::Die(int sides) { // assert(sides >= 1);
    srand( time(NULL) ); // initialize the random number generator
    numberOfSides = sides;
    currentRoll = rand() % sides + 1;
} // end Die constructor

void Die::roll() { // rolls the die
    currentRoll = rand() % numberOfSides + 1;
} // end roll

int Die::getRoll() const { // returns the value of the current roll
    return currentRoll;
} // end getRoll

bool Die::operator==( const Die & RHSDie) const{
    return (this->currentRoll == RHSDie.currentRoll);
} // end operator==

Die & Die::operator++( ) { // end operator++
    if (currentRoll < numberOfSides) {
        currentRoll = currentRoll + 1;
    } // end if
    return *this;
}

ostream & operator<<( ostream & output, const Die & die) { // enables "cout << a << b << c;"
    output << die.currentRoll;
    return output;
} // end operator<<

istream & operator>>( istream & input, Die & die){
    input >> die.currentRoll;
    input >> die.numberOfSides;
    assert(die.currentRoll >= 1 && die.currentRoll <= die.numberOfSides);
    return input;
} // end operator>>