Events this week

Schlumberger Info Session
Date: Mon., Feb 8
Time: 5:30 pm
Location: HENN Rm 201

Finding a Summer Job or Internship Info Session
Date: Wed., Feb 10
Time: 12 pm
Location: X836

Masters of Digital Media Program Info Session
Date: Thurs., Feb 11
Time: 12:30 – 1:30 pm
Location: DMP 201

Reminder: Co-op Deadline
Date: Fri., Feb 12
Submit application to Fiona at Rm X241 by 4:30 pm

Olympics Opening Ceremonies
Date: Fri., Feb 12
Time: 6:00pm

Administrivia

• Lecture slides (day by day) are on the web:
  • http://www.cs.ubc.ca/~norm/211/2009W2
• Assignment #2 is due this week
  • Due Thursday February 11, 10:00pm
Where are we?

• We are looking at:
  • The Java Collections Framework
• We’ve just looked at loops using iterators
  • Over collections of fixed type
  • Over collections of generic type

For-Each Loop

• Java provides a special type of for loop (called for-each loop) which can be used with any collection
• Example: Another version of the method that prints out all the elements in a collection of strings:

```java
public static void print( Collection<String> col )
{
    for (String str : col)
    {
        System.out.println(str);
    }
}
```
• Write the generic version using a for-each loop
For-Each Loop and Collection Modification

- A for-each loop cannot modify the collection over which the loop iterates
  - if this rule is violated Java throws a ConcurrentModificationException
- The following method that removes all accounts with low balance IS WRONG:
  ```java
  public static void removeBelow(
      Collection<Account> accounts, double limit ) {
    for ( Account acc : accounts ) {
      if ( acc.getBalance() < limit )
        accounts.remove(acc);   // WRONG
    }
  }
  ```

Iterators and Collection Modification

- A collection cannot be modified during the time an iterator iterates over it, unless it is done through the iterator remove() method
  - if this rule is violated Java throws a ConcurrentModificationException
- The following method IS WRONG:
  ```java
  public static void removeBelow(
      Collection<Account> accounts, double limit ) {
    Iterator<Account> itr = accounts.iterator();
    while( itr.hasNext() ){
      Account acc = itr.next();
      if( acc.getBalance() < threshold )
        accounts.remove(acc);   // WRONG
    }
  }
  ```
Iterators and Collection Modification (cont’d)

• The following is the correct code for this method:

```java
public static void removeBelow(
    Collection<Account> accounts, double limit)
{
    Iterator<Account> itr = accounts.iterator();
    while( itr.hasNext() ){
        Account acc = itr.next();
        if( acc.getBalance() < threshold )
            itr.remove();
    }
}
```

Exercises

1. Write a public static method that accepts a collection of type `Collection<String>` as an argument and removes all objects in collection `c` that satisfy the test `boolean test(String)`

2. Write a generic public static method that removes duplicates from a collection.

3. Based on Project 20.1 (Text)
   Implement a class `Polynomial` that represents a polynomial such as:
   
   \[ p(x) = 5x^{10} + 9x^7 - x - 10 \]
   
   Store a polynomial as a list of terms that contains the coefficient and power of x. So, for example, the polynomial above will be represented as:
   
   (5,10), (9,7), (-1,1), (-10,0)
   
   Supply methods to evaluate the polynomial for a given value of x.
Time Complexity of Algorithms

You are expected to:

- use big-O notation to categorize an algorithm as constant, linear, quadratic, logarithmic and exponential time
- given two or more algorithms, rank them in terms of their time efficiency

Complexity of Algorithms

- In the coming lectures, we’ll be discussing different implementations of collections and comparing them with respect to certain operations.
- We need to have a good way to define the performance of an algorithm (or a piece of code).
- In this section, we examine a means of analyzing the performance of an algorithm. Usually we are interested in the algorithm’s
  - *time complexity*: time taken for an algorithm to run
  - *space complexity*: amount of memory required by it
- In this course we mainly interested in time complexity.
Time Complexity

• One approach to determining an algorithm's time complexity would be to count the number of CPU cycles (or CPU time) it takes the algorithm to perform its operation
  • tedious and not a very practical approach
  • depends on the machine
• Instead we will count the number of simple statements (or steps) which are executed by the algorithm for a given input value n (time will be a function of n).
• By simple statement we mean a statement whose running time does not depend on n:
  • an assignment (without function calls)
  • a comparison between variables, etc.
For instance, a loop that executes n times would contribute n times the number of steps of the body

Big-O Notation

• We are not interested in an exact count of steps. Instead we want to know how fast the time grows as n grows. So, we use the following approximation
• Definition: Let T and f be functions of n. We say that T is $O(f(n))$ (pronounced “big-O f(n)” or “O f(n)”) if:
  $$ T(n) \leq c f(n) $$
  for any $n > n_0$
where $c$ and $n_0$ are constants.
• Example: Suppose that T is the time taken for an algorithm to sort an array of length n and that:
  $$ T(n) \leq c n^2 $$
  for all n
then we say that the algorithm is $O(n^2)$. 
Example 1

```java
int count = 0;
int sum = 0;
while( count < N )
{
    sum += count;
    count++;
}
System.out.print( "The sum is:");
System.out.println( sum );
```

- The time complexity of it depends on N.
- So
  \[ T(N) = 2 + 3N + 2 = 4 + 3N \]
- and
  \[ T(N) \leq 4N + 3N \leq 7N \leq cN \quad (c=7) \]
- Therefore
  \[ T(N) \text{ is } \mathcal{O}(n) \]

Linear Algorithms

- Algorithms like the previous one are called "linear algorithms"
- This means that the time taken to execute the algorithm \( T(n) \) for large values of \( n \) is \( \mathcal{O}(n) \)
- It also means that the time for the algorithm grows linearly as \( n \) grows
- Let's suppose that we double \( n \). How does this affect the time taken to execute the algorithm?
Example 2

```c
int count = 0;
int sum = 0;
while( count < N )
{
    int index = 0;
    while( index < N )
    {
        sum += index * count;
        index++;
    }
    count++;
}
```

• \( T(N) = 2 + N(2 + N(3) + 1) \)
  \[ = 2 + N(3N+3) \]
  \[ = 3N^2 + 3N + 2 \]

• Then
  \[ T(N) \leq 3N^2 + 3N^2 + 2N^2 \]
  or
  \[ T(N) \leq 8N^2 \]

• Therefore:
  \( T(N) \) is \( O(N^2) \)

Quadratic Algorithms and More

• Algorithms like the previous one are called "quadratic algorithms"

• This means that the time taken to execute the algorithm for large values of \( n \) is \( O(n^2) \)

• It also means that the time for the algorithm grows by \( n^2 \) as \( n \) grows

• In general we are interested in the following algorithm types:

<table>
<thead>
<tr>
<th>Algorithm Type</th>
<th>( T(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Linear</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Polynomial</td>
<td>( O(n^k) )</td>
</tr>
<tr>
<td>Exponential</td>
<td>( O(k^n) )</td>
</tr>
</tbody>
</table>

where \( k \) is an int constant.