Announcements

- Program 1 due Two Weeks!
- Lab 3-4 required completion this week
  - First web-cat submission

Material

- Evaluation of “efficiency”
  - Time Analysis
  - Big Oh notation
  - Complexity Classes
  - Linked Structures
Implementation Efficiency

• How should we measure efficiency in implementation?
  • What affects efficiency?
  • What should be measured? Speed? Memory?
  • Why should/shouldn’t we use execution speed?

• Remember, to manage anything, we need to measure it. So, how do we compare two implementations?

• ArrayBag vs. LinkedBag
  • Which is more efficient?
Analysis Metrics

Program Running (Execution) Time Factors
- Machine Speed (not just CPU speed)
- Programming Language and Implementation
- Compiler Code Generation (optimization)
- Input Data Size
- Time Complexity of Algorithm
  † Number of executed statements: $T(n)$
  † Function of the size of the input (termed $n$)
Implications

- Compiler code generation & processor speed differences are too great to be used as a basis for impartial algorithm comparisons.
- Overall system load may cause inconsistent timing results, even if the same compiler and hardware are used.
- Hardware characteristics, such as the amount of physical memory and the speed of virtual memory, can dominate timing results.
- In any case, those factors are irrelevant to the complexity of the algorithm.
• Consider unit task in relation to number of data elements
  • LC use dish washing example: 30 seconds per dish for washing, 30 seconds per dish drying
  • time = n * (30 sec wash + 30 sec dry)
  • But if not careful, see book for math
  • time = 15(n*n) + 45n seconds

• First algorithm more efficient, overall

• Growth function - shows relation between size of problem (n) and value to be optimized
When attempting an exact time analysis:

1. We assume an arbitrary time unit.
2. Running of each of the following operations takes time $T(1)$:
   a) assignment operations
   b) I/O operations
   c) Boolean operations
   d) arithmetic operations
   e) function return
3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.
4. Loop execution time is the time for the loop setup (initialization & setup) + the sum, (over the number of times the loop is executed), of the body time + time for the loop check and update operations. (Loop setup will include the termination check on pre-test loops.)

† Always assume that the loop executes the maximum number of iterations possible

5. Running time of a function call is $T(1)$ for setup + the time for any parameter calculations + the time required for the execution of the function body.

Non-executable statements, (e.g., declarations), are not counted. Only executable statements are analyzed.
Simple Summation

- Given:

```c
for (i = 0; i < n-1; i++) {
    for (j = 0; j < i; j++) {
        array[i][j] = 0;
    }
}
```

- Rules 4 and 2a: time 1 before loop
- Rules 4 & 2a: time 1 on each iteration of outer loop
- Rules 4 & 2a: time 1 on each iteration of inner loop
- Rules 4, 2c and 2d: time 3 on each iteration of outer loop
- Rules 4, 2c and 2d: time 2 (on each iteration of inner loop)
- Rule 2a: time 1 on each pass of inner loop
Simple Summation

• Continued:

† Outer loop will execute n -1 times -- this is the external sum.

† Inner loop will execute i times -- this is the internal sum.

So, the total time $T(n)$ is given by:

$$T(n) = 3 + \sum_{i=1}^{n-1} \left( 5 + \sum_{j=1}^{i} 3 \right)$$

Summation from applying Rule 4 to outer loop

Summation from applying Rule 4 to inner loop
Summation Formulas

Let N \geq 0, let A, B, and C be constants, and let f and g be any functions. Then:

\[ \sum_{k=1}^{N} Cf(k) = C \sum_{k=1}^{N} f(k) \]

S1: factor out constant

\[ \sum_{k=1}^{N} (f(k) \pm g(k)) = \sum_{k=1}^{N} f(k) \pm \sum_{k=1}^{N} g(k) \]

S2: separate summed terms

\[ \sum_{k=1}^{N} C = NC \]

S3: sum of constant

\[ \sum_{k=1}^{N} k = \frac{N(N+1)}{2} \]

S4: sum of k

\[ \sum_{k=1}^{N} k^2 = \frac{N(N+1)(2N+1)}{6} \]

S5: sum of k squared
The previous summation formulas can be used to evaluate the expressions obtained when analyzing the complexity of an algorithm:

\[ T(n) = 3 + \sum_{i=1}^{n-1} \left( 5 + \sum_{j=1}^{i} 3 \right) \]

\[ = 3 + \sum_{i=1}^{n-1} (5 + 3i) \]

\[ = 3 + \sum_{i=1}^{n-1} 5 + 3 \sum_{i=1}^{n-1} i \]

\[ = 3 + 5(n-1) + 3 \frac{(n-1)n}{2} \]

\[ = \frac{3}{2} n^2 + \frac{7}{2} n - 2 \]

From analysis on previous slide.

Apply S3 to the inner sum.

Apply S2 and S1 to the outer sum.

Apply S3 to the first sum, and apply S4 to the second sum.

Simplify and combine terms.
Order of Magnitude

Function Estimation:
Given algorithm that takes time: \( f(n) = 3n^2 + 5n + 100 \)

Graphically:

Algebraically:

If \( n \geq 10 \) then \( n^2 \geq 100 \)
If \( n \geq 5 \) then \( n^2 \geq 5n \)
Therefore, if \( n \geq 10 \) then:
\[
f(n) = 3n^2 + 5n + 100 < 3n^2 + n^2 + n^2 = 5n^2
\]

So \( 5n^2 \) forms an “upper bound” on \( f(n) \) if \( n \) is 10 or larger (asymptotic bound). In other words, \( f(n) \) doesn't grow any faster than \( 5n^2 \) “in the long run”.
Growth Function

- Interested in dominant term in expression
  \[ t(n) = 15 \times (n \times n) + 45 \times n \]

- We ignore the 45\(n\) term, and growth function becomes
  \[ t(n) = 15 \times (n \times n) \]

- Asymptotic complexity - how function behaves as \(n\) increases
- Order of an algorithm is characterized as the asymptotic complexity
- Order is measured in Big Oh, written \(O(n)\)
Big-O notation is used to express the asymptotic growth rate of a function.

- Formally suppose that $f(n)$ and $g(n)$ are functions of $n$. Then we say that $f(n)$ is in $O(g(n))$ provided that there are constants $C > 0$ and $N > 0$ such that for all $n > N$ then $f(n) \leq Cg(n)$.

- We often say that “$f$ is big-O of $g$” or that “$f$ is in $O(g)$”.
- By the definition above, demonstrating that a function $f$ is big-O of a function $g$ requires that we find specific constants $C$ and $N$ for which the inequality holds (and show that the inequality does, in fact, hold).
Example: from the previous slide, if $n > 10$ then

$$f(n) = 3n^2 + 5n + 100 \leq 5n^2$$

So, by the definition above, $f(n)$ is in $O(n^2)$.

Note that $5n^2 \leq 9n^2$ (for all $n$), so we could also conclude that $f(n)$ is $O(9n^2)$. Usually, we’re interested in the tightest (smallest) upper bound, so we’d prefer $5n^2$ to $9n^2$. 
The analysis given on slide 8 of these notes is typical of Big-O analysis, and is somewhat tricky. In order to simplify our work, we state the following theorems about Big-O:

1. Assume that f(n) is a function of n and that K is an arbitrary constant.
2. Thm 1: K is $O(1)$
3. Thm 2: A polynomial is $O($ the term containing the highest power of $n$ $)$
4. Thm 3: $Kf(n)$ is $O(f(n))$ [i.e., constant coefficients can be dropped]
5. Thm 4: For any base $b$, $\log_b(n)$ is $O(\log(n))$.
6. Thm 5: In general, $f(n)$ is big-O of the dominant term of $f(n)$, where “dominant” may usually be determined by the following list:
Complexity Classes

• Dominant Term List:

  constants
  \( \log_b(n) \) [always log base 2 if no base is shown]
  \( n \)
  \( n \log_b(n) \)
  \( n^2 \)
  \( n \) to higher powers
  \( 2^n \)
  \( 3^n \)
  larger constants to the \( n \)-th power
  \( n! \) [n factorial]
  \( n^n \)
Complexity Growth

Common Growth Curves

- \( \log n \)
- \( n \)
- \( n \log n \)
- \( n^2 \)
- \( n^3 \)
- \( 2^n \)
- \( 10^n \)

Order \( n^2 \) & less
Complexity Growth

Observations

• Algorithms with Order > n^2 require FAR more time than algorithms with Order n^2 or less, even for fairly small input sizes.

• For small n, there’s not much practical difference between Order n^2 and order log n.
## Typical Orders

<table>
<thead>
<tr>
<th>Order</th>
<th>Common name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>constant time</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>linear time</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>square time</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>logarithmic time</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>exponential time</td>
</tr>
</tbody>
</table>
Practical Complexity Classes

Low-order Curves

- \( n \) (input size)
- \( \log n \)
- \( n^2 \)
- \( n \log n \)
Observations

Even for moderately small input sizes, Order $n^2$ algorithms will require FAR more time than Order $n \log(n)$ algorithms.

**constants of proportionality**, (coefficients & lesser terms), have very little effect for large values of $n$ (between complexity classes).

Large problems with Order $> n \log(n)$ cannot practically be executed

For $n = 1000$ (medium problems) $n^2$ algorithms can still be used
Big-O Simple Summation

- Recall

```java
for (i = 0; i < n-1; i++) {
    for (j = 0; j < i; j++) {
        array[i][j] = 0;
    }
}
```

had a total time complexity $T(n)$ is given by:

$$T(n) = 3 + \sum_{i=1}^{n-1} \left( 5 + \sum_{j=1}^{i} 3 \right)$$

...and that $T(n)$ reduced to:

$$T(n) = \frac{3}{2} n^2 + \frac{7}{2} n - 3$$

Now by Theorem 2:

$$T(n) \in O\left(\frac{3}{2} n^2\right)$$

...and then by Theorem 3:

$$T(n) \in O\left(n^2\right)$$

So we’d say that the given code fragment is of order $n^2$ complexity.

Of course, this involved some unnecessary work. Big-O analysis provides a gross indication of the complexity of an algorithm, and that can be obtained without first doing an exact analysis.
Big-O Analysis Rules

When attempting an approximate big-O time analysis:

1. We assume an arbitrary time unit.
2. Running of each following type of statement takes time T(1): [omitting arithmetic operators]
   a) assignment statement
   b) I/O statement
   c) Boolean expression evaluation
   d) function return
3. Running time of a selection statement (if, switch) is T(1) for the condition evaluation + the maximum of the running times for the individual clauses in the selection.
4. Loop execution time is the time for the loop setup (initialization & setup) + the sum, over the number of times the loop is executed, of the body time + time for the loop check and update operations.
   - Always assume that the loop executes the maximum number of iterations possible
5. Running time of a function call is T(1) for function setup + the time required for the execution of the function body.
6. Running time of a sequence of statements is the largest time of any statement in the sequence.

indicates from the rules for exact analysis stated earlier.
Big-O Array Summation

• Order Analysis
Analysis will deal with the statements labeled a .. h; executable statements only.

Inner Loop: Sum from 0..i (or 1..i+1) of the loop body.
Body Rule 6: Maximum of \{loop condition, e, f\}
Rule 2: loop condition, e, and f each take 1
So, the inner loop body takes time \(O(1)\) and so the inner loop is

\[
O\left(\sum_{j=1}^{i+1} 1\right) = O(i + 1) = O(i)
\]

```c
void sumItoN(int[] ray, int n) {
    int i, j, t;
    i = 0;                  // a
    while (i <= n) {        // b (outer loop)
        j = t = 0;           // c
        while (j <= i) {     // d (inner loop)
            t = t + ray[j];   // e
            j++;              // f
        }
        ray[i] = t;          // g
        i++;                 // h
    }
}
```
Big-O Array Summation

**Order Analysis**

Outer Loop: Sum from 0..n (or 1..n+1) of the loop body.

Body Rule 6: Maximum of \{loop condition, c, inner loop, g, h\}

Rule 2: loop condition, c, g, and h each take time 1

So, the outer loop body takes time \(O(i)\) and so the outer loop is:

\[
O\left(\sum_{i=1}^{n+1} i\right) = O\left(\frac{(n+1)(n+2)}{2}\right) = O\left(\frac{1}{2}n^2 + \frac{3}{2}n + 1\right) = O(n^2)
\]

Finally by Rule 6, the big-O complexity of the function is the maximum of the outer loop and statement 1, which is \(O(1)\); so the function is \(O(n^2)\).

```c
void sumItoN(int[] ray, int n) {
    int i, j, t;
    i = 0;                      // a
    while (i <= n) {            // b (outer loop)
        j = t = 0;             // c
        while (j <= i) {       // d (inner loop)
            t = t + ray[j];    // e
            j++;              // f
        }
        ray[i] = t;            // g
        i++;                    // h
    }
}
```
Array Summation (exact count)

• Time Analysis

Analysis will deal with statements labeled a .. h; **executable** statements only.

Inner Loop: Sum from 0..i (or 1..i+1) of the loop body.

Body Rule 4: Sum of \{ loop condition, e, f \}

Rule 2: loop condition and f each take time 1; e takes time 2

The inner loop body takes time 4 & thus time for the inner loop is:

\[
\sum_{j=1}^{i+1} 4 = 4(i + 1)
\]
Array Summation (exact count)

Time Analysis

Outer Loop: Sum from 0..n (or 1..n+1) of the loop body.

Body Rule 6: Sum of { loop condition, c, inner loop condition, inner loop, g, h}

Rule 2: loop condition, g, and h each take time 1; c takes time 2

So, the outer loop body takes \(4(i+1) + 6\) or \(4i + 10\), and thus the outer loop plus statement a is:

\[
T(n) = 2 + \sum_{i=1}^{n+1} (4i + 10) = 2 + 4 \frac{(n+1)(n+2)}{2} + 10(n+1) = 2n^2 + 16n + 12
\]
Assume:

1 day $\approx 100,000$ sec. $\approx 10^5$ sec.  (actually 86,400)
Input size $n = 10^6$
A computer that executes $1,000,000$ Inst/sec (Java statement instructions)

**Algorithm Complexity Class Comparison**

<table>
<thead>
<tr>
<th>Order: $n^2$</th>
<th>Order: $n \log_2 n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(10^6)^2$ Instructions</td>
<td>$10^6 \log_2 10^6$ Instructions</td>
</tr>
<tr>
<td>$10^{12}$ Instructions</td>
<td>$20 (10^6) = 2 (10^7)$</td>
</tr>
<tr>
<td>$10^{12} / 10^6$ secs to run</td>
<td>$2 (10^7) / 10^6$ secs to run</td>
</tr>
<tr>
<td>$10^6$ secs to run</td>
<td>$10^6 / 10^5$ days to run</td>
</tr>
<tr>
<td>$10^6 / 10^5$ days to run</td>
<td>20 sec to run</td>
</tr>
<tr>
<td>10 days to run</td>
<td>10 days to run</td>
</tr>
</tbody>
</table>
Faster Processors

Does the fact that hardware is always becoming faster hardware mean that algorithm complexity doesn’t really matter?

Suppose we could obtain a machine that was capable of executing 10 times as many instructions per second (so roughly 10 times faster than the machine hypothesized on the previous slide).

How long would the order \( n^2 \) algorithm take on this machine with an input size of \( 10^6 \)?

<table>
<thead>
<tr>
<th>Order: ( n^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td># instructions:</td>
</tr>
<tr>
<td># seconds to run:</td>
</tr>
<tr>
<td># days to run:</td>
</tr>
</tbody>
</table>

Impressed?

You shouldn’t be. That’s still 1 day versus 20 seconds if an algorithm of order \( n \log(n) \) were used.

What about 100 times faster hardware? 2.4 hours.
Examples

• Consider the bag class that we have been discussing, what is the time complexity of these routines
  • Insert an element
  • Find an element
  • Remove an element
  • Remove all elements
  • Compare two bags
  • Remove all duplicates
  • Intersection of two bags
Limitations of Arrays

- Limitations of arrays
  - Fixed size
    - Unused storage early on
    - Inefficient when growing
    - Gets worse over time, each grow operation is slower
  - Linear

- Solution
  - Linked structures
    - Avoids unused space, avoids copy operation
    - Drawback: sequential access
Terminology

- **Self-referential objects**

  ```java
class Student {
    String name;
    Student next;
    ... typical accessors here
  }
  ```

- **Collection built of linked objects**

  ```java
  Student a = new Student("Joe");
  Student b = new Student("Mae");
  Student c = new Student("Pat");
  a.setNext(b);
  b.setNext(c);
  // a points to // collection
  ```

- **Problems?**
Problems with approach

• Linked representation is mixed with data representation

```java
class Student {
    // ...student data goes here...
    Student next; // this is for the Linked representation
}
```

• Better solution, separate the node and self-reference from the data

```java
class Node {
    Object data; // data goes here
    Node next; // link to self goes here
}
```
Improved approach

- Now the node objects are generic and can be used by all collections
- Node is independent of data stored
- So, what does a linked representation of a collection looks like?

- A collection holds a link to the first element in the list

```java
class MyCollection {
    Node head; // first element
    int count;
    ...
}
```
Visualizing linked lists

- Overtime, the collection looks like this
Add

• Different ways to add objects
  • at the beginning of the sequence
  • in the middle of the sequence
  • at the end of the sequence

• Linked structures are sequential
  • to get to end, we need to visit all parts
Add at head

Add to front of list:

//========================================
//  Adds the specified element to the bag
//========================================

public void addToFront (Object element)
{
    Node node = new Node(Element.clone(), head);
    head = node;
    count++;
}
Add in middle

- Add to center of list

```java
// Adds specified element to the bag after position
public void addAfter (Object element, int position)
{
    if (position >= 0) && (position < count))
    {
        Node link = head;
        for (int p = 0; (p < position); p++)
            link = link.getNext();
        Node node = new Node(Element.clone(), link.getNext());
        link.setNext(node);
        count++;
    }
}
```
Add at tail of list

// Adds specified element to the bag at the end

public void addToRear (Object element)
{
    if (count == 0)
    {
        addToFront(element); return;
    }

    Node link = head;
    for (int p = 0; (p < count-1); p++)
    {
        link = link.getNext();
    }
    Node node = new Node(Element.clone(), null);
    link.setNext(node);
    count++;
}
Using dummy Node

- **Buffer Nodes**
  - Lists which have an extra node at the beginning and end of the list.
  - Dummy nodes do not store data, (only used to store links).
  - Eliminates special cases for adding and removing.
  - For adding, no special case needed for inserting before the head.
  - For removing, no special case needed for removing the last node.
Managing LinkedBag

• Classes
  
  class LinkedBag implements BagADT {
    private Node head;
    private int count;

    ...
  }

• Constructor
  
  public LinkedBag {
    head = null;
    count = 0;
  }

• Where should Node be declared? Public or private?
Internal Classes

- Classes defined **inside** other classes
  - Access privilege apply (public, private, etc)

- LinkedBag uses Node internally
  - Nowhere in the interface does it return a Node object
  - Thus, define class internally

- Another excellent way to do encapsulation
public class LinkedBag {
    private class Node {
        private Node next;
        private Object data;
        Node(Object d, Node n) {
            data = d.clone();
            next = n;
        }

        Object getData() { return data; }
        void setData(Object d) {
            data = d.clone();
        }

        Node getNext() { return next; }
        void setNext(Node n) { next = n; }
    }
    private Node head;
    private int count;
    public LinkedBag() { ... }
}

*Internal class definition, no class outside of LinkedBag can access it (because it is private)*
public void add(Object data) {
    Node newElement = new Node(data, head);
    head = newElement;
}

Adding elements
## Analysis

- How does the linked structures compare with array based implementations?

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>O(1), could be O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>addAll</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>contains</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>