Lecture 8: Array Lists
Continued

CS 62
Spring 2019
William Devanny & Alexandra Papoutsaki
Order of Magnitude

• **Definition:** We say that $f(n)$ is $O(g(n))$ iff there exist two constants $C$ and $k$ such that
  \[ |f(n)| \leq C |g(n)|, \text{for all } n > k. \]

• Used to measure time and space complexity of algorithms on data structures of size $n$.

• **Examples:**
  • $2n + 1$ is $O(n)$
  • $n^3 - n^2 + 83$ is $O(n^3)$
  • $2^n + n^2$ is $O(2^n)$
Order of Magnitude

• Most common are:
  • $O(1)$ - constant
  • $O(\log n)$ – logarithmic
  • $O(n)$ – linear
  • $O(n^2)$ – quadratic
  • $O(n^c)$ – polynomial
  • $O(c^n)$ – exponential
  • $O(n!)$ – factorial

• Growth:
  • $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(2^n), O(n!)$
Complexity

Figure 5.2  Near-origin details of common curves. Compare with Figure 5.3.
Figure 5.2 Near-origin details of common curves. Compare with Figure 5.3.

Figure 5.3 Long-range trends of common curves. Compare with Figure 5.2.
Properties of $O(f(n))$

1. Ignore constant factors
   - Ex. $2n$ is $O(n)$  
     $1,000,000n$ is $O(n)$

2. Ignore lower order terms
   - Ex. $n + \log n$ is $O(n)$  
     $2^n + n^2$ is $O(2^n)$
   - Why?
     - Interested in how runtime grows for large inputs!
     - The constant factors and lower order terms will be out scaled.
Comparing Orders of Magnitude

- Suppose we have the operations with complexities given and that a problem of size $n$ takes time $t$.
- How long would it take if we increase size of problem?

<table>
<thead>
<tr>
<th>Problem Size:</th>
<th>$10n$</th>
<th>$100n$</th>
<th>$1000n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(\log n)$</td>
<td>$3+t$</td>
<td>$7+t$</td>
<td>$10+t$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>$10t$</td>
<td>$100t$</td>
<td>$1000t$</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>$&gt;10t$</td>
<td>$&gt;100t$</td>
<td>$&gt;1000t$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>$100t$</td>
<td>$10000t$</td>
<td>$1,000,000t$</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>$\sim t^{10}$</td>
<td>$\sim t^{100}$</td>
<td>$\sim t^{1000}$</td>
</tr>
</tbody>
</table>
Rule of thumb
Adding to ArrayList

• Suppose there are $n$ elements in ArrayList and you want to add one more. What is the cost of this operation?

• If enough space (size<capacity):
  • Add to end is $O(1)$
  • Add to beginning is $O(n)$

• If not space:
  • What is the cost of ensureCapacity?
  • $O(n)$ because $n$ elements in array
Amortized Time Analysis

As the Arraylist increases in size, the doubling happens half as often but costs twice as much
Amortized Time Analysis

We will use the aggregate method – the simplest method for amortized time analysis
Others: accounting (banker’s) and potential (physicist’s).
Think of it as an average. For a total of \( n \) operations: \( O(\text{total cost}) / (\text{number of operations}) \)
(in this case additions)

\[
O(\text{total cost of operations}) = \sum \text{cost of insertions} + \sum \text{cost of copying}
\]

Total cost of insertions: \( n \) of them, each \( O(1) \) cost, therefore \( nO(1) = O(n) \)
Total cost of copying during the \( n \) insertions: \( 1 + 2 + 2^2 + \cdots + 2^{\log n} \leq 2n \) which is \( O(n) \)
\[
O(\text{total cost}) = O(n) + O(n) = O(n)
\]
Amortized time = \( O(n)/n = O(1) \) but “lumpy”
What if we only increase the capacity by 1 element each time?

- Adding \( n \) elements one at a time to end
  - Total cost of \( n \) insertions: \( 1 + 2 + 3 + \cdots + (n - 1) = \frac{n(n - 1)}{2} \)
  - Total cost of \( O(n^2) \)
- Average cost of each is \( O(n) \)
ArrayList Operations

• Best case:
  • $O(1)$: size(), isEmpty(), get(int i), set(int i, E elem), remove(), add()

• Worst case:
  • $O(1)$: size(), isEmpty(), get(int i), set(int i, E elem)
  • $O(n)$: remove, add()

• add() runs in amortized constant time: adding $n$ elements requires $O(n)$ time.