The Efficiency of Algorithms

Chapter 4
Contents

• Motivation
• Measuring an Algorithm’s Efficiency
  ▪ Counting Basic Operations
  ▪ Best, Worst, and Average Cases
• Big Oh Notation
  ▪ The Complexities of Program Constructs
Contents

• Picturing Efficiency
• The Efficiency of Implementations of an ADT
  ▪ An Array-Based Implementation
  ▪ A Linked Implementation
  ▪ Comparing the Implementations
Objectives

• Assess efficiency of given algorithm
  ▪ Use case: understand scalability to large inputs
• Compare expected execution times of two methods
  ▪ Given efficiencies of algorithms
  ▪ Use case: choose an algorithm given multiple choices
  ▪ Example: President Obama and Computer Science
http://www.youtube.com/watch?v=k4RRi_ntQc8
Motivation

• Contrast algorithms

Figure 4-1 Three algorithms for computing the sum $1 + 2 + \ldots + n$ for an integer $n > 0$
Motivation

- Java code for algorithms

```java
// Algorithm A
long sum = 0;
for (long i = 1; i <= n; i++)
    sum = sum + i;
System.out.println(sum);
```

```java
// Algorithm B
sum = 0;
for (long i = 1; i <= n; i++)
{
    for (long j = 1; j <= i; j++)
        sum = sum + 1;
} // end for
System.out.println(sum);
```

```java
// Algorithm C
sum = n * (n + 1) / 2;
System.out.println(sum);
```

- Even a simple program can be inefficient
Measuring an Algorithm’s Efficiency

- Complexity
  - Space and time requirements
  - Worst Case, Average Case, Best Case
  - Classes:
    - constant, logarithmic, linear, quadratic, …, NP

- Other issues for best solution
  - Generality of algorithm
  - Programming effort
  - Problem size – number of items program will handle
  - Growth-rate function
Counting Basic Operations

Figure 4-2 The number of basic operations required by the algorithms in Figure 4-1

<table>
<thead>
<tr>
<th></th>
<th>Algorithm A</th>
<th>Algorithm B</th>
<th>Algorithm C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additions</td>
<td>$n$</td>
<td>$n(n + 1) / 2$</td>
<td>1</td>
</tr>
<tr>
<td>Multiplications</td>
<td>$n$</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Divisions</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Total basic operations</td>
<td>$n$</td>
<td>$(n^2 + n) / 2$</td>
<td>3</td>
</tr>
</tbody>
</table>
Figure 4-3 The number of basic operations required by the algorithms in Figure 4-1 as a function of input size \( (n) \).
FIGURE 4-4 Typical growth-rate functions evaluated at increasing values of input size (n)

<table>
<thead>
<tr>
<th>n</th>
<th>log(log n)</th>
<th>log n</th>
<th>log^2 n</th>
<th>n</th>
<th>n log n</th>
<th>n^2</th>
<th>n^3</th>
<th>2^n</th>
<th>n!</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2</td>
<td>3</td>
<td>11</td>
<td>10</td>
<td>33</td>
<td>10^2</td>
<td>10^3</td>
<td>10^3</td>
<td>10^5</td>
</tr>
<tr>
<td>10^2</td>
<td>3</td>
<td>7</td>
<td>44</td>
<td>100</td>
<td>664</td>
<td>10^4</td>
<td>10^6</td>
<td>10^30</td>
<td>10^94</td>
</tr>
<tr>
<td>10^3</td>
<td>3</td>
<td>10</td>
<td>99</td>
<td>1000</td>
<td>9966</td>
<td>10^6</td>
<td>10^9</td>
<td>10^301</td>
<td>10^1435</td>
</tr>
<tr>
<td>10^4</td>
<td>4</td>
<td>13</td>
<td>177</td>
<td>10,000</td>
<td>132,877</td>
<td>10^8</td>
<td>10^12</td>
<td>10^3010</td>
<td>10^19,335</td>
</tr>
<tr>
<td>10^5</td>
<td>4</td>
<td>17</td>
<td>276</td>
<td>100,000</td>
<td>1,660,964</td>
<td>10^10</td>
<td>10^15</td>
<td>10^30,103</td>
<td>10^243,338</td>
</tr>
<tr>
<td>10^6</td>
<td>4</td>
<td>20</td>
<td>397</td>
<td>1,000,000</td>
<td>19,931,569</td>
<td>10^12</td>
<td>10^18</td>
<td>10^301,030</td>
<td>10^2,933,369</td>
</tr>
</tbody>
</table>

Copyright ©2012 by Pearson Education, Inc. All rights reserved
Best, Worst, and Average Cases

• Some algorithms depend only on size of data set
• Other algorithms depend on nature of the data
  ▪ Consider: Search for an element in a list
  ▪ Best case search when item at beginning
  ▪ Worst case when item at end
  ▪ Average case somewhere between
Big Oh Notation

• Notation to describe algorithm complexity

• Definition
  A function $f(n)$ is of order at most $g(n)$ that is, $f(n)$ is $O(g(n))$—if:
  • A positive real number $c$ and positive integer $N$ exist such that:
    \[ f(n) \leq c \cdot g(n) \]
    for all $n \geq N$.

• That is, $c \cdot g(n)$ is an upper bound on $f(n)$ when $n$ is sufficiently large.
Figure 4-5 An illustration of the definition of Big Oh: $f(n)$ is $O(g(n))$
Figure 4-6 An $O(n)$ algorithm

```plaintext
for i = 1 to n
    sum = sum + i
```

1 2 3 ... n $O(n)$
Figure 4-7 An \( \mathcal{O}(n^2) \) algorithm

\begin{verbatim}
for i = 1 to n
    \{ for j = 1 to i
        sum = sum + 1
    \}
\end{verbatim}

\[
O(1 + 2 + \ldots + n) = \mathcal{O}(n^2)
\]
Figure 4-8 Another $O(n^2)$ algorithm
Classes:

- **constant**: search using hash table (map)
- **logarithmic**: binary search
- **linear**: brute-force search
- **n*log(n)**: best sorting algorithms (e.g., merge sort)
- **quadratic**: bubble sort
- **NP**: Travelling salesperson problem
Figure 4-10: The time required to process one million items by algorithms of various orders at the rate of one million operations per second.

<table>
<thead>
<tr>
<th>Growth-Rate Function $g$</th>
<th>$g(10^6) / 10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>log $n$</td>
<td>0.0000199 seconds</td>
</tr>
<tr>
<td>$n$</td>
<td>1 second</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>19.9 seconds</td>
</tr>
<tr>
<td>$n^2$</td>
<td>11.6 days</td>
</tr>
<tr>
<td>$n^3$</td>
<td>31,709.8 years</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$10^{301,016}$ years</td>
</tr>
</tbody>
</table>
Array Based Implementation

Ex.: What is the worst case complexity of add() and getIndex()?

Choices: (a) constant, (b) logarithmic, (c.) linear, (d) n*\log(n), (e) quadratic, (f) NP

```java
public boolean add(T newEntry)
{
    boolean result = true;
    if (isFull())
    {
        result = false;
    }
    else
    {
        // assertion: result is true here
        bag[numberOfEntries] = newEntry;
        numberOfEntries++;
    } // end if
    return result;
} // end add

private int getIndexOf(T anEntry)
{
    int where = -1;
    boolean found = false;
    for (int index = 0; !found && (index < numberOfEntries); index++)
    {
        if (anEntry.equals(bag[index]))
        {
            found = true;
            where = index;
        } // end if
    } // end for
    return where;
} // end getIndexOf
```
Array Based Implementation

• **Exercise:** What is the worst case complexity of `add()`?

```java
public boolean add(T newEntry)
{
    boolean result = true;
    if (isFull())
    {
        result = false;
    }
    else
    {
        // assertion: result is true here
        bag[numberOfEntries] = newEntry;
        numberOfEntries++;
    }
    // end if

    return result;
} // end add
```

• Adding an entry is an O(1) method
Array Based Implementation

• **Exercise:** What is the worst case complexity of add()?

```java
private int getIndexOf(T anEntry) {
    int where = -1;
    boolean found = false;
    for (int index = 0; !found && (index < numberOfEntries); index++) {
        if (anEntry.equals(bag[index])) {
            found = true;
            where = index;
        } // end if
    } // end for
    return where;
} // end getIndexOf
```

• Searching for an entry, O(1) best case, O(n) worst or average case
• Thus an O(n) method overall
Big Oh Identities

- $O(k \, g(n)) = O(g(n))$ for a constant $k$
- $O(g_1(n)) + O(g_2(n)) = O(g_1(n) + g_2(n))$
- $O(g_1(n)) \times O(g_2(n)) = O(g_1(n) \times g_2(n))$
- $O(g_1(n) + g_2(n) + \ldots + g_m(n)) = O(\max(g_1(n), g_2(n), \ldots, g_m(n)))$
- $O(\max(g_1(n), g_2(n), \ldots, g_m(n))) = \max(O(g_1(n)), O(g_2(n)), \ldots, O(g_m(n)))$
A *Linked* Implementation

**Exercise:** What is the worst case complexity of `add()`?

```java
class Node {
    T entry;
    Node next;
}

class LinkedContainer {
    Node firstNode;
    int numberOfEntries;

    public boolean add(T newEntry) { // OutOfMemoryError possible
        Node newNode = new Node(newEntry);
        newNode.next = firstNode; // make new node reference rest of chain
        firstNode = newNode; // (firstNode is null if chain is empty)
        numberOfEntries++;
        return true;
    }
}
```

**Adding an entry is an O(1) method**
A Linked Implementation

Exercise: What is the worst case complexity of add()?

```java
public boolean contains(T anEntry) {
    boolean found = false;
    Node currentNode = firstNode;
    while (!found && (currentNode != null)) {
        if (anEntry.equals(currentNode.data))
            found = true;
        else
            currentNode = currentNode.next;
    } // end while

    return found;
} // end contains
```

- Searching for a given entry, O(1) best case, O(n) worst case
- O(n) overall

Copyright ©2012 by Pearson Education, Inc. All rights reserved
Figure 4-11 The time efficiencies of the ADT bag operations for two implementations, expressed in Big Oh notation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Fixed-Size Array</th>
<th>Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(newEntry)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove(anEntry)</td>
<td>O(1), O(n), O(n)</td>
<td>O(1), O(n), O(n)</td>
</tr>
<tr>
<td>clear()</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>getFrequencyOf(anEntry)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>contains(anEntry)</td>
<td>O(1), O(n), O(n)</td>
<td>O(1), O(n), O(n)</td>
</tr>
<tr>
<td>toArray()</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>getCurrentSize(), isEmpty(), isFull()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
End

Chapter 4