Efficiency of Algorithms

Some solutions are better than others!
Comparing Two Versions of Selection Sort

Same idea, different implementation. Is it important?
Comparing Different Sorting Algorithms

Both algorithms sort correctly. Is one better than the other?

```java
public void selectionSort (int[] data) {
    int maxNum;        // max integer
    int maxIndex;      // index of max integer
    int i, j;
    for (j = data.length - 1; j > 0; j--) {
        maxIndex = 0;
        maxNum = data[0];
        for (i = 1; i <= j; i++)
            if (data[i] > maxNum) {
                maxNum = data[i];
                maxIndex = i;
            }
        swap(data, maxIndex, j);
    }
}

void swap (int[] data, int i, int j) {
    // exchanges the contents
    // of data[i] and data[j]
    int temp = data[i];
    data[i] = data[j];
    data[j] = temp;
}
```

```java
public void insertionSort(int[] theArray, int n) {
    for (int unsorted = 1; unsorted < n; ++unsorted) {
        int nextItem = theArray[unsorted];
        int loc = unsorted;
        while ((loc > 0) &&
               (theArray[loc-1] > nextItem)){
            // shift theArray[loc-1] to right
            theArray[loc--] = theArray[loc-1];
        }
        // insert nextItem into sorted region
        theArray[loc] = nextItem;
    }
}
```
“Towers of Hanoi” game
Comparing any Two Algorithms

Algorithms solving two different problems. Is one problem harder than the other?

public void solveTowers(int n, char source, char dest, char spare) {
    if (n==1)
        System.out.println("Move top disk from " +
        source + " to " + dest);
    else {
        solveTowers(n-1, source, spare, dest);
        solveTowers(1, source, dest, spare);
        solveTowers(n-1, spare, dest, source);
    }
}

public void selectionSort (int[] data) {
    int maxNum;  // max integer
    int maxIndex;  // index of max integer
    int i, j;
    for (j = data.length - 1; j > 0; j--)
    {
        maxIndex = 0;
        maxNum = data[0];
        for (i = 1; i <= j; i++)
        {
            if (data[i] > maxNum) {
                maxNum = data[i];
                maxIndex = i;
            }
        }
        swap(data, maxIndex, j);
    }
}

void swap (int[] data, int i, int j) {
    // exchanges the contents
    // of data[i] and data[j]
    int temp = data[i];
    data[i] = data[j];
    data[j] = temp;
}
Determining the Efficiency of Algorithms

- What is an efficient algorithm?
  - Faster is better
  - How do you measure time? Wall clock? Computer clock?
  - Using less space is better
  - And if you need to get data in/out of main memory it takes time

- Algorithm efficiency should be independent of
  - Specific implementations and coding tricks (programming language, control statements)
  - Specific Computers (hardware chip, OS, clock speed)

- But size of data should matter
  - But particular set of data should not matter

- Analysis of algorithms
  - Is a major field that provides math tools for evaluating the efficiency of various algorithmic solutions to problems
Execution Time of Algorithms

• Count an algorithm's operations to assess its efficiency
• An algorithm’s execution time is related to the number of operations it requires in a worst case scenario
• Examples of worst case scenarios
  • **Searching** a linked list
    • Operations: about as many as elements
  • **Selection sort**
    • Operations: as we go through the array to select the next minimum, we traverse most of the array again
  • The Towers of Hanoi
    • Operations: to solve an instance of n disks, we need to solve 2 instances of n-1 disks…
• We could consider an average case scenario, but it’s harder
Single-statement Execution

A statement that the computer can execute in one or a few (fixed number of) instructions, we count it as 1 step:

\[ O(1) = \text{"order of one"} \]

This includes arithmetic operations, logical operations, assignments, but not necessarily function calls, recursive steps, etc.

For example:

```c
// code with O(1) steps
int i = 100;
if (data[i] > maxNum) {
    maxNum = data[i];
    maxIndex = i;
}
```

Each of these statements is \( O(1) \).

Thus the overall order of this piece of code is 
\[ k \times O(1) = O(1) \]
where \( k \) is the number of individual statements.
Analyzing Single Loop Execution

- Need to determine how often a set of statements gets executed to determine the order of an algorithm
- To analyze loop execution, first determine the order of the body of the loop, and then multiply that by the number of times the loop will execute

```java
// n = numbers.length is the size of the array
for(int i = 0; i < n; i++) {
    if (numbers[i] == 0) {
        count++;
    }
}
```

The loop executes n times, and the body of the loop is O(1). Thus the overall order is O(n) * O(1) = O(n)
Analyzing Nested Loop Execution

- When loops are nested, we must multiply the complexity of the outer loop by the complexity of the inner loop.

```java
// n = numbers.length is the size of the array
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        if (numbers[i] + numbers[j] == 0) {
            count++;
        }
    }
}
```

Both the inner and outer loops have complexity of $O(n)$.

When multiplied together, the order becomes $O(n^2)$. 

Determining the order of a recursive algorithm

- determine the **order of the recursion** (# of times recursive definition is followed) and **multiply it by the order of the body** of the recursive method

**Example:** Consider the recursive method to compute the product of integers from 1 to some $n > 1$

```java
public int fact (int n) {
    int result;
    if (n == 1)
        result = 1;
    else
        result = n * fact(n-1);
    return result;
}
```

- **Size of the problem is** $n$, the number of values to be multiplied
- **Operation of interest is the multiplication operation**
- **The body of the method performs one multiplication and therefore is** $O(1)$
- Each time the recursive method is invoked, $n$ is decreased by 1, thus...
  - **...the recursive method is called** $n$ times
- **The order of the entire algorithm is** $O(n)$
Algorithm Growth Rates

- An algorithm’s time requirements can be measured as a function of the problem size `n`.

- Function’s growth rate enables the comparison between algorithms.
  - Examples
    - **Searching** a sorted array requires time proportional to `lg n`.
    - **Searching** a list requires time proportional to `n`.
    - Selection **sort** requires time proportional to `n^2`.

- Notation: **Big-Oh** aka “order of”
  - Searching a sorted array requires time `O(lg n)`.
  - Searching a list requires time `O(n)`.
  - Selection sort requires time `O(n^2)`.

- Algorithm efficiency is typically a concern for large problems only (as `n` grows...).
Comparison of Growth Rates

<table>
<thead>
<tr>
<th>Function</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\log_2 n$</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>16</td>
<td>19</td>
</tr>
<tr>
<td>n</td>
<td>$10^2$</td>
<td>$10^3$</td>
<td>$10^4$</td>
<td>$10^5$</td>
<td>$10^6$</td>
<td></td>
</tr>
<tr>
<td>n * $\log_2 n$</td>
<td>30</td>
<td>664</td>
<td>9,965</td>
<td>$10^5$</td>
<td>$10^6$</td>
<td>$10^7$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$10^2$</td>
<td>$10^4$</td>
<td>$10^6$</td>
<td>$10^8$</td>
<td>$10^{10}$</td>
<td>$10^{12}$</td>
</tr>
<tr>
<td>$n^3$</td>
<td>$10^3$</td>
<td>$10^6$</td>
<td>$10^9$</td>
<td>$10^{12}$</td>
<td>$10^{15}$</td>
<td>$10^{18}$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$10^3$</td>
<td>$10^{30}$</td>
<td>$10^{301}$</td>
<td>$10^{3,010}$</td>
<td>$10^{30,103}$</td>
<td>$10^{301,030}$</td>
</tr>
</tbody>
</table>

Time units needed to solve problem of size $n$. The graph shows the growth of time units as a function of problem size for different growth rates.
Order-of-Magnitude Analysis and “Big-Oh”

• Definition of the order of an algorithm:
  Algorithm $A$ is order $f(n)$, denoted $A = O(f(n))$, if constants $k$ and $n_0$ exist such that $A$ requires no more than $k \times f(n)$ time units to solve a problem of size $n \geq n_0$

• Growth-rate function
  – A mathematical function used to specify an algorithm’s order in terms of the size of the problem

• “Big-Oh” notation
  – A notation that uses the capital letter $O$ to specify an algorithm’s order
  – Example: $O(n)$, $O(n^2 \times \log n)$, $O(n^4)$, in general, $O(f(n))$
Order-of-Magnitude Analysis and “Big-Oh”

- Order of growth of some common functions

\[ O(1) < O(\log_2 n) < O(n) < O(n \cdot \log_2 n) < O(n^2) < O(n^3) < O(2^n) \]

**Note** that < is not the arithmetic “less than” but means “smaller order”

Where would you place:
- \( O(n^6) \)?
- \( O((\log_2 n)^2) \)?
Order-of-Magnitude Analysis and “Big-Oh”

Properties of growth-rate functions

- Summing orders is dominated by the larger order
  \[ O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max\{f(n),g(n)\}) \]
  Therefore you can ignore low-order terms
  \[ O(n^2 + n) = O(n^2) \]

- Multiplying orders means multiplying terms
  \[ O(f(n)) \times O(g(n)) = O(f(n) \times g(n)) \]
  Therefore you can ignore multiplicative constants
  \[ O(12 \times n^2) = O(n^2) \]
What is the running time?

```java
/**
 * Adds a CD to the collection of n CDs, increasing the
 * size of the collection if necessary
 */

public void addCD (String title, String artist, double cost, 
                   int tracks) {
    if (count == collection.length)
        increaseSize();
    collection[count] = new CD(title, artist, cost, tracks);
    totalCost += cost;
    count++;
}
```
/**
 * Increases the capacity of the collection by creating a larger array and copying
 */

private void increaseSize(){
    CD[] temp = new CD[collection.length * 2];

    for (int cd = 0; cd < collection.length; cd++){
        temp[cd] = collection[cd];
    }

    collection = temp;
}
Analysis of Stack and Queue Implementations

- All operations for a stack (push, pop, peek, etc.) are $O( )$
- Almost all operations for a queue are $O( )$
- The only exception is the dequeue operation for the ArrayQueue implementation – the shifting of elements makes it $O( )$
- The dequeue operation for the CircularArrayQueue is $O( )$ because of the ability to eliminate the shifting of elements
- Both stacks and queues can be implemented very efficiently
Worst Case and Average Case Analyses

• An algorithm can require different times to solve different problems of the same size

  – **Worst-case analysis**
    • A determination of the *maximum* amount of time that an algorithm requires to solve problems of size n
    • Big-O uses worst-case analysis

  – **Average-case analysis**
    • A determination of the *average* amount of time that an algorithm requires to solve problems of size n
Keeping your Perspective

- Throughout the course of an analysis, keep in mind that you are interested only in significant differences in efficiency.

- When choosing an ADT’s implementation, consider how frequently particular ADT operations occur in an application.

- Some seldom-used but critical operations must be efficient.

- If the problem size is always small, you can probably ignore an algorithm’s efficiency.

- Weigh the trade-offs between an algorithm’s time requirements and its memory requirements.

- Compare algorithms for both style and efficiency.

- Order-of-magnitude analysis focuses on large problems.