Efficiency of Algorithms

- Memory keeps getting bigger, processors faster (Moore's Law), both cheaper, so why care about efficiency?
  - Problems get bigger, too
  - Big problems may overwhelm the best resources
  - Speed still a concern (how long do you want to wait for Goggle?)

Measuring Algorithm Efficiency

- How to compare two algorithms?
- Could code both and run the programs, but results may be confounded by
  - Language choice, machine choice
  - Coding style
  - Input data choice
- Want to measure essential efficiency of algorithm without these factors
Complexity Analysis

- Assess the efficiency of the algorithm, not a specific program implementation
  - Important for choosing best solution
  - Focus is efficiency of algorithm, not saving compute time with coding tricks
  - Concerned with savings as problem grows in size – does the algorithm scale up

Measuring Algorithms

- Time
  - Number of steps executed
  - Assignments, tests, arithmetic, etc.

- Space
  - Amount of memory needed
  - Variables, arrays, lists, dynamic objects

- How do these measures change with problem's size
**RAM Complexity Model**

- Estimate time-space needs of computation assuming basic machine (von Neumann architecture)
  - Processor performs operations on data
  - Data in random access memory with equal time access to all cells
  - Bus transfers data between processor and memory

**RAM Complexity Model**

- Times for machine operations
  - tfetch, tstore
  - tadd, tsub, tmult, tdiv, tcompare, ...
  - tbranch, treturn

- Examples
  - \( x = y; \)  // tfetch + tstore
  - \( x = y + z; \)  // 2*tfetch + tadd + tstore
factorial Time Analysis

# of executions                                             Execution Time

factorial(n)                                              
1                                                      tfetch+tstore
1                                                      tfetch+tstore
n-1                                                         2*tfetch+tcomp+tbr
n-1                                                        tfetch+tadd+tstore
n-1                                                        2*tfetch+tmult+tstore
1                                                      tfetch+treturn

Simplify Model

- Constants for each instruction set

  factorial(n)
  
  result = 1;                                               c_0
  for (i=2;                                                 c_0
    i <= n;                                                 (n-1)*c_1
    ++i)                                                    (n-1)*c_2
  result *= i;                                             (n-1)*c_3
  return result;                                          c_4

- Total time: \((2*c_0-c_1-c_2-c_3+c_4)+n*(c_1+c_2+c_3)\)
  or just: \(k_0 + n*k_1\)
Execution Time

- Instruction times depend on machine, but constant on a particular machine
- Constant times not so important (a faster machine will come along)
- Algorithm time can be expressed as a function of the input size: $t(n)$
  - Number of primitive steps needed for input $n$

Complexity Measure

- Algorithms have conditional steps (selection statements) and iteration (loops) that depend on data values
  - Even for same size inputs, data values may change way algorithm executes
  - Possible different time complexity for same input size
- Need to look at best, average, and worst cases
  - Define time complexity to be worst case
  - Sometimes practical considerations still mean a poor worst case algorithm is a better choice
isMember Example

- Given \( n \) objects held in array \( a \),
  Return true if object \( x \) is in the array,
  false otherwise

```java
boolean isMember(Object x, Object [] a) {
    boolean result = false;            // Default is not found
    for (int i=0; i < a.length; ++i) {  // Loop through all
        if (x.equals(a[i])) {
            result = true;                // If found, return true
            break;                        // terminate loop
        }
    }
    return result;
}
```

isMember Time Analysis

```java
boolean isMember(Object x, Object [] a) {
    boolean result = false;            // Default is not found
    for (int i=0; i < a.length; ++i) {  // Loop through all
        if (x.equals(a[i])) {
            result = true;                // If found, return true
            break;                        // terminate loop
        }
    }
    return result;
}

worst case: \( 2 \cdot c \cdot n + 3 \cdot c \)
```
Time Analysis

- May focus on certain operations for time estimate (active operations)
- For isMember
  - Number of calls to equals
    - best case: 1 (when x is first element)
    - worst case: n (when x is last or not found)
  - Number of assignments to result
    - best case: 1 (when x is not found)
    - worst case: 2 (when x is found)

Counting Inversions

- Given an integer array, return number of times $i^{th}$ element is greater than $i+1^{th}$

```java
int adjacentInversions(int[] a) {
    int count = 0; // Default is none
    for (int i = 0; i < a.length - 1; ++i) {
        if (a[i] > a[i + 1]) {
            ++count;
        }
    }
    return count;
}
```

worst case: $3 \cdot c \cdot n + c$


**min max Example**

- Given array of numbers, 
  Return the minimum and maximum

```java
int[] findMinMax(int[] a) { // Assume at least one value
    int[] result = new int[2];
    result[0] = result[1] = a[0]; // Default min, max
    for (int i=1; i < a.length; ++i) { // Loop through all
        if (a[i] < result[0])
            result[0] = a[i]; // New minimum
        if (a[i] > result[1])
            result[1] = a[i]; // New maximum
    }
    return result;
}
```

**findMinMax Analysis**

- Number of comparisons
  - All cases: $2n - 2$  
    (we are looping for all but first)
  - What if second "if" changed to "else if"

- Number of assignments to result
  - Best case: 2 (min, max same and first value)
  - Worst case: $n$ (increasing values)
Another minmax Algorithm

```java
int[] findMinMax(int[] a) { // Assume length even, > 1
    int[] res = new int[2];
    int[] lres = new int[2];
    if (a[0] > a[1]) // Local min, max of first two
        { res[0] = a[1]; res[1] = a[0]; }
    else
        { res[0] = a[0]; res[1] = a[1]; }
    for (int i=2; i < a.length; i+=2) { // Loop by pairs
        if (a[i] > a[i+1]) // Local min, max of next two
            { lres[0] = a[i+1]; lres[1] = a[i]; }
        else
            { lres[0] = a[i]; lres[1] = a[i+1]; }
        if (lres[0] < res[0]) res[0] = lres[0]; // update
    }
    return res; // worst case: 1 + 3(n-2)/2 comparisons
}
```

Algorithm Growth Rates

- Measures expressed in terms of input size for best, worst cases
  - Functions can be complicated formulas
  - How to compare for significant differences?
- For small input size, we don't care
- For constant factors of difference, we don't care as much
  - Speed of machine addresses this
- Major concern is how quickly it grows with the input size
Algorithm Growth Rates

For example, suppose
  - Algorithm A is proportional to n
  - Algorithm B is proportional to n^2
  - Clearly, algorithm A is a better choice if n > 1

What if
  - Algorithm A takes time \( 5 \cdot n \)
  - Algorithm B takes time \( n^2 / 5 \)
  - Which is better?

Big O notation allows us to quantify

---

**Figure 9.1**
Time requirements as a function of the problem size \( n \)
Big O Notation

- Measure of order of magnitude of growth
  - Let \( n \) represent size of problem (e.g., number of elements in an array)
  - Suppose algorithm has complexity function \( f(n) \)
- Definition: The algorithm of complexity \( f(n) \) is said to be of order \( g(n) \), denoted \( O(g(n)) \) if there is a constant \( c \) such that \( f(n) \leq c \cdot g(n) \) for all \( n > n_0 \)

- Big O is a meaningful approximation
  - \( g(n) \) bounds \( f(n) \) from above for large enough \( n \)
  - We say "\( f(n) \) is Big-O of \( g(n) \)"

Order-of-Magnitude Analysis and Big O Notation

<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</td>
<td>10^2</td>
<td>10^3</td>
<td>10^4</td>
<td>10^5</td>
<td>10^6</td>
</tr>
<tr>
<td>( n \times \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^{10}</td>
<td>10^{30}</td>
<td>10^{301}</td>
<td>10^{3,010}</td>
<td>10^{301,030}</td>
</tr>
</tbody>
</table>
Order-of-Magnitude Analysis and Big O Notation

Figure 9.3b
A comparison of growth-rate functions: b) in graphical form

Showing Big O Complexity

- Prove \( n^2 + n \) is \( O(n^2) \)
  - Let \( c=2 \), let \( n_0=1 \), then \( cn_0^2 = 2 \geq n_0^2 + n_0 = 2 \)
  - For \( n > 1 \), \( n^2 = n^2 + n^2 > n^2 + n \)
- \( 2n^3 + 10n^2 + 1000 \) is \( O(n^3) \)
  - Let \( c=1012 \) and let \( n_0=1 \)
- Rule 1: Big-O of a polynomial is the highest power of \( n \) after eliminating coefficients
- Rule 2: An exponential in \( n \) dominates all polynomial terms
  - \( 2^n + n^4 \) is \( O(2^n) \)
Big O Analysis

- Relation of common functions:
  \[ O(1) < O(\log_2 n) < O(n) < O(n \cdot \log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!) \]

- Properties of Big O
  
  - May ignore low-order terms
  - May ignore multiplicative constant in high-order term
  
  \[ O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max[f,g]) \]
  \[ O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n)) \]

- So the two algorithms for minmax are both \( O(n) \)
  
  - Second algorithm is faster, but
  - the "shape" of the growth of complexity is the same

Other Complexity Measures

- Omega: we say \( f(n) \) is \( \Omega(g(n)) \) if there is a constant \( c \) such that \( f(n) \geq c \cdot g(n) \) for all \( n > n_0 \)
  
  - \( g(n) \) bounds \( f(n) \) from below for large \( n \)

- Theta: we say \( f(n) \) is \( \Theta(g(n)) \) if and only if \( f(n) \) is \( O(g(n)) \) and \( \Omega(g(n)) \)
  
  - \( g(n) \) bounds \( f(n) \) from above and below for large \( n \)
  
  - Typically this is a simpler function that closely approximates growth – a "tight" Big O bound
  
  - Moving \( g \) up a constant factor, \( g \) dominates \( f \)
  
  - Moving \( g \) down a constant factor, \( f \) dominates \( g \)
Iterative Complexity

- A loop may have sequences, conditions and selection of procedure calls (which are other algorithms)
  - How do we determine Big-O for an iterative solution, i.e., how to combine complexity measurements?
- Sequential
  - Just add complexity of components in sequence (which means find the max complexity)
- Repetition
  - Count repetitions (another complexity measurement) and multiply by complexity of block repeated
- Combine these to determine complexity of overall algorithm
  - Keeping worst case in mind

Iterative Complexity Example

- Suppose we have methods do-x, do-y, and do-z with complexities $O(1)$, $O(n)$, and $O(n \cdot \log n)$ respectively

```c
void do-it(int [] a) {
    for (int i=0; i<n; ++i) {
        j<n; ++j) {
            if ((i%2)==0) {
                for (int j=0; j<n; ++j) {
                    for (int i=0; i<n; ++i) {
                        k<n; ++k) {
                            do-x();
                            do-y(a); }
                        do-z(a); }
                    do-z(a); }
                do-y(a); }
            do-x(); }
        do-y(a); }
    do-it(int [] a) {
        for (int i=0; i<n; ++i) {
            j<n; ++j) {
                if ((i%2)==0) {
                    for (int j=0; j<n; ++j) {
                        for (int i=0; i<n; ++i) {
                            k<n; ++k) {
                                do-x();
                                do-y(a); }
                            do-z(a); }
                        do-z(a); }
                    do-y(a); }
                do-x(); }
            do-y(a); }
        do-it(int [] a) {
            for (int i=0; i<n; ++i) {
                j<n; ++j) {
                    if ((i%2)==0) {
                        for (int j=0; j<n; ++j) {
                            do-x();
                            do-y(a); }
                        do-z(a); }
                    do-z(a); }
                do-y(a); }
            do-y(a); }
```

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Week 2  CIS 212  Spring 2006  27

Week 2  CIS 212  Spring 2006  28
Selection in a Loop

- Must consider how often selection is made
- Assume \( f(n) \) is \( O(n^2) \) and \( g(n) \) is \( O(n \cdot \log n) \)

```c
O(n) for (int i=0; i<n; ++i) {
    if ((i%2 == 0)
        x += f(n);
    else
        x += g(n);
}
```

```c
O(n/2) for (int i=0; i<n/2; ++i) {
    if (i < 10)
        x += f(n);
    else
        x += g(n);
}
```

```c
O(10) for (int i=0; i<10; ++i) {
    x += f(n);
    else
        x += g(n);
}
```

Set Algorithms

- If you go through all \( n \) elements in a set a constant number of times and do a constant amount of work per element each time, then the algorithm is \( O(n) \).
- If you go through \( O(f(n)) \) elements in a set \( O(g(n)) \) number of times and do \( O(h(n)) \) work per element each time, then the algorithm is \( O(f(n) \cdot g(n) \cdot h(n)) \).
- Recall isMember algorithm
  - Worst case and average case are \( O(n) \)
  - Best case was \( O(1) \)
  - Best case – element located in first fixed number of positions
  - Self organizing lists and arrays – when element is accessed, move it closer to front
    - Assumes most requests are about a few elements
Set Operation Complexity

- Using an array for the elements of a set

```java
interface SetI {
    boolean isEmpty(); O(1)
    int size(); O(1)
    boolean isMember(int val); O(n)
    void addElement(int val) O(n)
    void deleteElement(int val); O(n)
}
```

*Improve to $O(\log n)$ by keeping elements in increasing order and using Binary Search*

Binary Search

- Array elements are kept in increasing order
- isMember(39) – how many steps to determine if the value 39 is in the array?

```
2  7  13  17  20  32  39  46  64  80
```

- Split problem in half each time, so worst case is least $k$ such that $2^k \geq n$
- Algorithm is $O(\log n)$
Binary Search Implementation

- Array is sorted by increasing value

```java
boolean isMember(Object x) {
    int low=0; high=a.length()-1; // Entire array
    while (low <= high) { // Segment to look at
        int mid = (low + high)/2; // Find midpoint
        if (x.equals(a[mid])) // Compare to mid element
            return true;        // if found, return true
        else if (x.lessThan(a[mid])) // Adjust range down
            high = mid – 1;   // to bottom half
        else                          // Adjust range up
            low = mid + 1;      // to top half
    }
    return false;                   // Not found
}
```

Some Useful Formulas

- Sum from $0$ to $n$  \[ \sum_{i=0}^{n} i = n(n+1)/2 = O(n^2) \]
- Sum from $0$ to $n^k$  \[ \sum_{i=0}^{n^k} i = n^k(n^k+1)/2 = O(n^{2k}) \]
- Sum of $k$ powers from $0$ to $n$  \[ \sum_{i=0}^{n} x^i = O(n^{k+1}) \]
- Sum of powers from $0$ to $n$ of $2$  \[ \sum_{i=0}^{n} 2^i = 2^{n+1} - 1 = O(2^n) \]
- Number of times one can divide $n$ by $m$ before reaching $l$  \[ \left| \log_m n \right| = O(\log n) \]