Analysis of algorithms is the area of computer science that provides tools for contrasting the efficiency of different methods of solution.

- Concerns with significant differences
- Clever coding tricks are often more that offset by reduced program readability
How To Do Comparison?

• Implement the algorithms in C# and run the programs
  – How are the algorithms coded?
  – What computer should you use?
  – What data should the programs use?
• Analyze algorithms independent of implementations
The Execution Time of Algorithms

• Count the number of basic operations of an algorithm
  – Read (get), write (put), compare, assignment, jump, arithmetic operations (increment, decrement, add, subtract, multiply, divide), shift, open, close, logical operations (not/complement, AND, OR, XOR), ...
The Execution Time of Algorithms

• Counting an algorithm’s operations

```
int sum = item[0];
int j = 1;
while (j < n) {
    sum += item[j];
    ++j;
}
```

<- 1 assignment
<- 1 assignment
<- n comparisons
<- n-1 plus/assignments
<- n-1 plus/assignments

Total: 3n operations
Algorithm Growth Rates

- Measure an algorithm’s time requirement as a function of the problem size
  - Number of elements in an array

Algorithm A requires $n^2/5$ time units
Algorithm B requires $5n$ time units

- Algorithm efficiency is a concern for large problems only
Common Growth-Rate Functions - I

(b)

Value of growth-rate function

\[ 2^n \]

\[ n^3 \]

\[ n^2 \]

\[ n \log_2 n \]

\[ n \]

\[ \log_2 n \]
# Common Growth-Rate Functions - II

<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>10²</td>
<td>10³</td>
<td>10⁴</td>
<td>10⁵</td>
<td>10⁶</td>
</tr>
<tr>
<td>n * \log_2 n</td>
<td>30</td>
<td>664</td>
<td>9,965</td>
<td>10⁵</td>
<td>10⁶</td>
<td>10⁷</td>
</tr>
<tr>
<td>n²</td>
<td>10²</td>
<td>10⁴</td>
<td>10⁶</td>
<td>10⁸</td>
<td>10¹⁰</td>
<td>10¹²</td>
</tr>
<tr>
<td>n³</td>
<td>10³</td>
<td>10⁶</td>
<td>10⁹</td>
<td>10¹²</td>
<td>10¹⁵</td>
<td>10¹⁸</td>
</tr>
<tr>
<td>2^n</td>
<td>10³</td>
<td>10³₀</td>
<td>10³₀¹</td>
<td>10³₀¹⁰</td>
<td>10³₀¹³</td>
<td>10³₀¹³₀³₀</td>
</tr>
</tbody>
</table>
Big O Notation

Algorithm A is order \( f(n) \)-denoted \( O(f(n)) \)-if constants \( k \) and \( n_0 \) exist such that A requires \( \leq k \times f(n) \) time units to solve a problem of size \( n \geq n_0 \)

- \( n^2/5 \)
  - \( O(n^2) \): \( k=1/5, n_0=0 \)
- \( 5 \times n \)
  - \( O(n) \): \( k=5, n_0=0 \)
More Examples

• How about \( n^2 - 3n + 10 \)?
  
  \(- O(n^2) \) if there exist \( k \) and \( n_0 \) such that
  
  • \( kn^2 \geq n^2 - 3n + 10 \) for all \( n \geq n_0 \)
  • \( 3n^2 \geq n^2 - 3n + 10 \) for all \( n \geq 2 \); so \( k=3, n_0=2 \)
Properties of big-Oh

- Ignore low-order terms
  - $O(n^3 + 4n^2 + 3n) = O(n^3)$
- Ignore multiplicative constant
  - $O(5n^3) = O(n^3)$
- Combine growth-rate functions
  - $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
Software Life Cycle
Software Life Cycle

Phase 1: Specification

• Answers “What do we build?”

• Define clearly all aspects of the problem
  – What is input (valid/invalid) data?
  – What assumptions are possible?
  – Are there special cases?
  – What enhancements are likely in the future?
Software Life Cycle
Phase 2: Design

- Divide into manageable parts - modules
- Specify each module’s purpose, assumptions, input, and output
- Develop algorithms
- Look for existing software components
Software Life Cycle

Phase 3: Risk Analysis

- Attempts to answer “What can go wrong, and how bad can it be?”
- Predict and manage what risks you can
- Risks to timetable, cost, human health, etc. should be taken into account
- Risks can greatly influence the direction of a project
Software Life Cycle

Phase 4: Verification

- Answers “Are the algorithms correct?”
- Some algorithms can be proven correct
  - assertion: condition at a certain point
  - invariant: condition that is always true
Software Life Cycle

Phase 5: Coding

- Translate the algorithms into a particular programming language
- Minor phase in the software life cycle
Software Life Cycle

Phase 6: Testing

- Answers “Did we build it correctly?”
- Try to make the software fail
- Develop as many test cases as possible
- Both a science and an art
Software Life Cycle

Phase 7: Refining a Solution

- Add bells and whistles
- Retest after any changes are made
Software Life Cycle
Phase 8: Production

- Distribute it
- Install it
Software Life Cycle

Phase 9: Maintenance

- Fix previously undiscovered bugs
- Add new features
- Enhance old features
- Generally the most costly phase (80% of total cost by some estimates)
Software Life Cycle

Documentation

- Performed extensively in every phase
- In modern software development, there are typically different people working on each phase of the software life cycle
- Novice programmers usually undervalue documentation
Continuing Effort

- Specification
- Design
- Risk Analysis
- Verification
- Coding
- Testing
- Refining
- Production
- Maintenance

Documentation