

Ch 1

The Phases of Software Development

- Specification of the task
- Design of the solution
- Coding of the solution
(implementation)
- Analysis of complexity and efficiency
- Testing and debugging
- Maintenance

Specification of the Task

- Algorithmic approach
 - Specify inputs, outputs (and processing).
 - Practice iterative functional decomposition.
 - Establish data flow.
- Object-oriented approach
 - Write a narrative describing the problem.
 - Nouns become classes.
 - Adjectives become properties or states.
 - Verbs become functions.
 - Adverbs indicate polymorphic functions.

Algorithmic Design Techniques

- Optimize subtask characteristics.
 - small
 - functionally cohesive
 - uncoupled: global variables are avoided
 - general: parameters used for specific behaviors
- Use existing functions.
- Provide for exceptional conditions.
- Document the pre/post condition contract in the prototype, not the definition.

Preconditions and Postconditions

- A precondition is a statement giving the condition that is required to be true when a function is called. The function is not guaranteed to perform as it should unless the condition is true.
- A postcondition is a statement describing what will be true when a function exits. If the function is correct and the precondition was true when the function was called, the function will complete and the postcondition will be true.

Prototype Documentation

```
double celsius_to_fahrenheit(double c);  
// Precondition: c is a Celsius temperature no less than  
// absolute zero (-273.16).  
// Postcondition: The return value is the temperature c  
// converted to Fahrenheit degrees.  
  
void setup_cout_fractions(int fraction_digits);  
// Precondition: fraction_digits is not negative.  
// Postcondition: All double or float numbers printed to  
// cout will now be rounded to the specified digits on  
// the right of the decimal.
```

Programming Tips

- Use the standard namespace and the newer header files (without the `.h` extension).
- Use declared constants:
 - `const char HEADING1[] = "Celsius";`
 - `const int WIDTH = 9;`
- Check preconditions with `assert()`.
- Use `EXIT_SUCCESS` or `EXIT_FAILURE` in `main()`.

For Example...

```
#include <iostream>    // Provides cout
#include <iomanip>     // Provides setw function
#include <cstdlib>    // Provides EXIT_SUCCESS
#include <cassert>    // Provides assert function
using namespace std;  // Provides for standard namespace

double celsius_to_fahrenheit(double c) {
    const double MINIMUM_CELSIUS = -273.16;
    assert(c >= MINIMUM_CELSIUS);
    return (9.0 / 5.0) * c + 32;
}

void setup_cout_fractions(int fraction_digits) {
    assert(fraction_digits >= 0);
    cout.precision(fraction_digits);
    cout.setf(ios::fixed, ios::floatfield);
    if (fraction_digits == 0)
        cout.unsetf(ios::showpoint);
    else
        cout.setf(ios::showpoint);
}

int main( ) {
    // ...
    return EXIT_SUCCESS;
}
```

Comparing Algorithms

Implementing and running algorithms is often impractical or unsatisfactory.

- Multiple implementations must be produced when only one will reach production.
- Because of implementation variances, relative qualities of competing algorithms may not be truly represented.
- The choice of empirical test cases might unfairly favor one algorithm.
- The winner may not fall within the resource budget.

Running Time: $T(n)$

Operation	Clock Cycles	
		<pre>for (int i = 0; i < n; i++) cout << i << endl;</pre>
assignment	a	<pre>for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) cout << i << " " << j << endl;</pre>
comparison	c	<pre>for (int i = 0; i < n; i++) for (int j = i; j < n; j++) cout << i << " " << j << endl;</pre>
increment	i	
insertion	o	<pre>int b = 2; for (int i = n; i > 0; i /= b) cout << i << endl;</pre>

Rules for Running Time

- The running time of consecutive statements is the sum of their running times.
- The running time of a conditional statement is the time of the test plus the time of the largest alternative.
- The running time for a loop is at most the running time of the statements inside the loop (including tests) times the number of iterations.
- The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of the loops.

Asymptotic Algorithm Analysis

- The time required to run an algorithm expressed as a function of n is written as $T(n)$
- $T(n)$ is in the set $O(f(n))$ if there are two positive constants c and n_0 such that

for all

- If the upper bound (highest growth rate) for an algorithm is $f(n)$ then it is in $O(f(n))$ in the worst case
- For *all* inputs that are large enough, the algorithm *always* executes in less time than

Costs Compared



■	$y = \frac{3}{2}x$
■	$y = \frac{1}{7}x^2 + 1$

Simplification Rules

- If $f(n)$ is in $\mathbf{O}(g(n))$ and $g(n)$ is in $\mathbf{O}(h(n))$, then $f(n)$ is in $\mathbf{O}(h(n))$.
- If $f(n)$ is in $\mathbf{O}(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $\mathbf{O}(g(n))$.
- If $f_1(n)$ is in $\mathbf{O}(g_1(n))$ and $f_2(n)$ is in $\mathbf{O}(g_2(n))$, then $f_1(n) + f_2(n)$ is in $\mathbf{O}(\max(g_1(n), g_2(n)))$.
- if $f_1(n)$ is in $\mathbf{O}(g_1(n))$ and $f_2(n)$ is in $\mathbf{O}(g_2(n))$, then $f_1(n)f_2(n)$ is in $\mathbf{O}(g_1(n)g_2(n))$.

Simplification Rules Simplified

- If some function is an upper bound for a cost (time) function, then any upperbound for that function is also an upper bound for that cost function.
- Multiplicative constants may be ignored.
- If two parts run in sequence, only the more expensive part need be considered.
- The total cost of a repeated action is its cost times the number of iterations.

The Examples, Again

```
for (int i = 0; i < n; i++)  
    cout << i << endl;
```

$O(n)$

```
for (int i = 0; i < n; i++)  
    for (int j = 0; j < n; j++)  
        cout << i << " " << j << endl;
```

$O(n^2)$

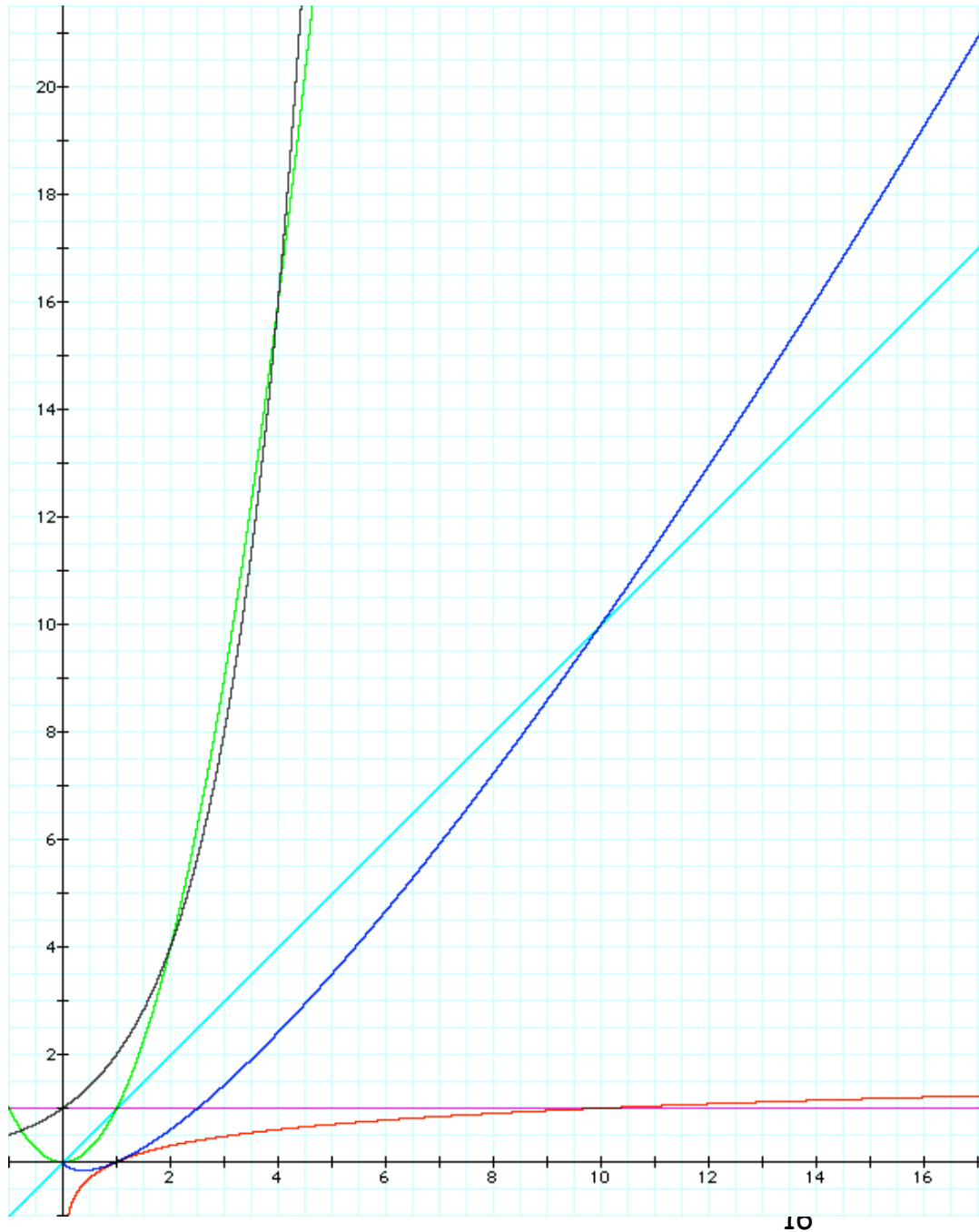
```
for (int i = 0; i < n; i++)  
    for (int j = i; j < n; j++)  
        cout << i << " " << j << endl;
```

$O(n^2)$

```
int b = 2;  
for (int i = n; i > 0; i /= b)  
    cout << i << endl;
```

$O(\log n)$

Common Functions



- $y = 1$
- $y = \log x$
- $y = x$
- $y = x \log x$
- $y = x^2$
- $y = 2^x$

Comparison of Running Times in Increasing Order

Program Testing

- Establish the mapping from inputs to outputs.
- Test the inputs that are most likely to cause errors.
- Determine and test boundary conditions.
- Fully exercise the code with a profiler.
 - Each statement should be executed at least once.
 - Provide data for skipping optional areas.

Debugging

- Don't change suspicious code without discovering known errors with test data.
- Run all test cases after making changes.
- Use a debugger to
 - set breakpoints
 - watch the values of expressions
 - alter variables during execution