

**Top-Down Design:** A solution method where the problem is broken down into smaller sub-problems, which in turn are broken down into smaller sub-problems, continuing until each sub-problem can be solved in a few steps.

(Also called: Divide and Conquer strategy)

**Problem:** An integer  $N$  is "perfect" if  $N$  is equal to the sum of the positive integers  $K$  such that  $K < N$  and  $K$  is a divisor of  $N$ .

Design an algorithm to determine if a given integer is "perfect".

The top-level sub-problems (or tasks) appear to be:

- I. Get the number which is to be tested.
- II. Determine the divisors of the number.
- III. Check if the divisors add up to the number.

Now we may consider each of these tasks separately, assuming the others will be taken care of...

Continuing our analysis:

- I. Get the number which is to be tested.
  - A. Output a prompt asking for the number.
  - B. Input the number; call it  $N$ .
  - C. Stop if  $N$  isn't positive.
- II. Determine the divisors of the number.
  - A. Set the divisor sum to 1. (Since 1 certainly divides  $N$ .)
  - B. Check each integer,  $D$ , from  $2 \dots N/2$  to see if  $D$  is a divisor of  $N$ .
- III. Check the results.
  - A. If the divisor sum equals  $N$  Then
    - i.  $N$  is perfect
  - Else
    - i.  $N$  is not perfect.

Now we may consider what details need to be added to clarify the process enough to make it an algorithm.

Continuing to add detail:

- I. Get the number which is to be tested.
  - A. Output: "Please enter a positive integer: "
  - B. Input the number; call it N.
  - C. If N isn't positive
    - i. Output: N " isn't perfect."
    - ii. STOP.
- II. Determine the divisors of the number.
  - A. Set the DivisorSum to 1. (Since 1 certainly divides N.)
  - B. Set D to 2.
  - C. While D is less than or equal to  $N / 2$ 
    - i. If D is a divisor of N
      - a. Add D to DivisorSum
    - ii. Add 1 to D.
- III. Check the results.
  - A. If DivisorSum equals N
    - i. Output: N " is perfect."
    - Else
    - i. Output: N " isn't perfect."
  - B. STOP.

The outline format used here is a fairly simple way to express an algorithm design.

It clarifies the ordering and grouping of actions, and doesn't require any special editing tools to create.

One of the most fundamental ideas in problem solving involves subdividing a problem into subproblems and solving each subproblem individually. Quite often, the attempt to solve a subproblem introduces new subproblems at a lower level. The subdivision of subproblems should continue until a subproblem can easily be solved. This is similar to the refining of an algorithm.

In large projects, subproblems may be assigned to different programmers, or teams, who will be given precise guidelines about how that subproblem fits into the overall solution, but who will have relatively complete freedom in attacking the particular subproblem at hand.

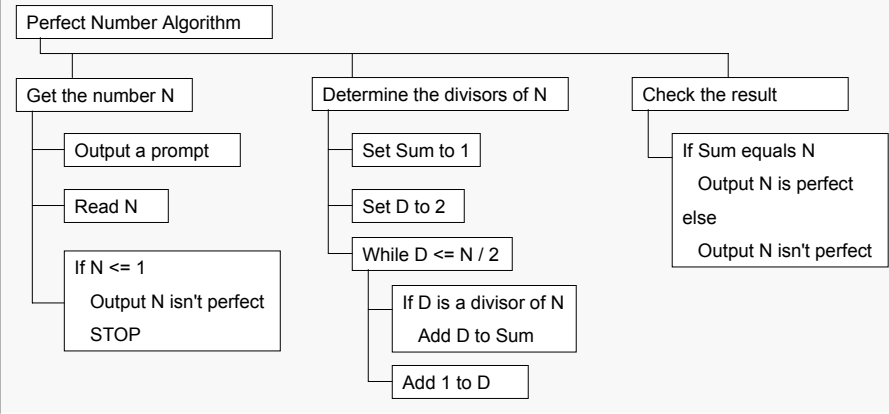
If there is a problem you can't solve, then there is an easier problem you can't solve: find it.

G. Pólya

Most common techniques:

1. Outlining (written decomposition)
2. Structure Chart (hierarchical tree diagram)
3. Pseudocode (mixture of English & programming constructs)

Structure chart for the Perfect Number Algorithm:



```

Comment Get the value for N
Output: "Please enter a positive integer: "
Input: N
If N <= 0 Then
    Output: N " isn't a perfect number."
    STOP
Endif

Comment Initialize sum of the divisors & divison
SumOfDivisors <-- 1
Candidate <-- 2

Comment Consider all possible divisors
While Candidate <= N / 2 Do
    R <-- N - (Truncate ( N / Candidate ) ) * Candidate
    If R = 0 Then
        SumOfDivisors <-- SumOfDivisors + Candidate
    Endif
    Candidate <-- Candidate + 1
Endwhile

Comment Check the results
If SumOfDivisors = N Then
    Output: N " is a perfect number."
Else
    Output: N " isn't a perfect number."
Endif
    
```

**Pseudocode is potentially clearer than either a structure chart or an outline.**

**Unfortunately there is no universal pseudocode notation.**

**Program proof:** code analysis to achieve a formal verification of the correctness of a program's output.

**Assertion:** statement of a relationship between a program's variables and data.

- Every program statement can be preceded and succeeded by an assertion, (termed the precondition and postcondition respectively).
- Program proofs involve applying deductive logic to show that the precondition and postcondition assertions are true for all program statements
- Due to practical time considerations assertions are made and proven only for sequences of statements.

Example:

```

// Pre: N >= 0, M > 0 ← precondition
while (M > 0)
    M = M / N;
// Post: N >= 0, M == 0 ← postcondition
0
    
```

loop invariants assertions that are true prior to loop execution and following each loop iteration.

Commonly composed of the boolean expression controlling the loop, its negation, ranges of loop variables, data structure states and file status. Derived by analyzing the setup and modifications of the loop variables.

Example:

```

. . .
// Integer multiplication by repeated addition.
// Calculates Multiplier * Multiplicand and stores result in
// Product.
// Pre: Multiplier >= 0, Multiplicand has been set
Product = 0;
Factor = Multiplier;

// Invariants: Product == Multiplicand * (Multiplier - Factor)
//             Factor is in the range [0, Multiplier]
while (Factor > 0) {
    Product = Product + Multiplicand;
    Factor = Factor - 1;
}
// Post: Product == Multiplier * Multiplicand
//       Factor == 0
. . .
    
```

The production of external program documentation prior to program implementation is a natural outgrowth of any good software development process.

External documentation describes the purpose of a program and its data. External documentation may also describe the overall architecture of the program design.

Internal documentation, in the form of in-line comments, as well as module documentation, is produced in conjunction with code development.

The production of good documentation should be viewed as a necessary and useful part of the software development process, not merely as an afterthought to satisfy requirements.

See the “Elements of Programming Style” in the appendices.

A **structure chart** is a documentation tool that enables the designer to keep track of the relationships among subproblems throughout this refinement process.

The structure chart gives a detailed description of the original problem in terms of subproblems. This top-down description will make it easier to write a program to solve the original problem.

We can solve each upper-level box in the structure chart by using a C++ **function**. A function is a grouping of statements into a single unit. This single unit can be invoked to solve its subproblem simply by activating it by means of a **function call**.

Since the structure chart gives a pictorial mechanism for visualizing the solution to a problem and each of the pieces of the structure chart corresponds to a function, the solution to a problem is obtained by correctly invoking a series of functions. In order to solve problems, we need only solve all the subproblems by means of functions. Thus a C++ program is a collection of functions that solve a series of subproblems. This is the essence of top-down design.

The top-down approach to program design involves *procedural abstraction*. This enables us to associate meaningful names with more complicated algorithm steps, reference these steps by name, and defer their implementation details until later.

Our programs are written using logically independent sections, similar to the manner in which we developed the solution algorithm (as logically independent steps).

Using logically independent program sections enables the programmer to hide the details of subproblem implementations from one another. This concept, *information hiding*, is critical in all stages of the software development process.

**Language syntax (compilation) errors:**

- Error is in the form of the statement: misspelled word, unmatched parenthesis, comma out of place, etc.
- Error is detected by the compiler (at compile time).
- Compiler cannot correct error, so no object file is generated.
- Compiler prints error messages, but usually continues to compile.

**Linking errors:**

- Error is typically in the form of the declaration or implementation or call of a function.
- Error may also result from including the wrong header file.
- Error is detected by the linker (after the compiler has produced an object file).
- Linker cannot correct error, so no executable file is generated.
- Linker prints error messages, but usually continues link analysis.

**Execution (runtime) errors:**

- Error occurs while the program is running, causing the program to "crash" (terminate abnormally) and usually producing an error message from the operating system.
- Error is frequently an illegal operation of some sort. The most common are arithmetic errors like an attempt to divide by zero, and access violations when the program tries to use some resource like a memory address that is not allocated to it.
- Program source code compiles and links without errors – no help there.
- Unfortunately, some operating systems do not reliably detect and respond to some kinds of execution errors. In that case, an incorrect program may appear to function correctly on one computer but not on another.

**Logic errors:**

- Error occurs while the program is running, causing the production of incorrect results, but not necessarily a runtime "crash".
- Program source code compiles and links without errors – no help there.
- Logic errors are detected by checking results computed by the program.
- The cause(s) of the error must be determined by a logical analysis of the error and the source code. This must be done by the developer. This is the hardest type of error to deal with.

