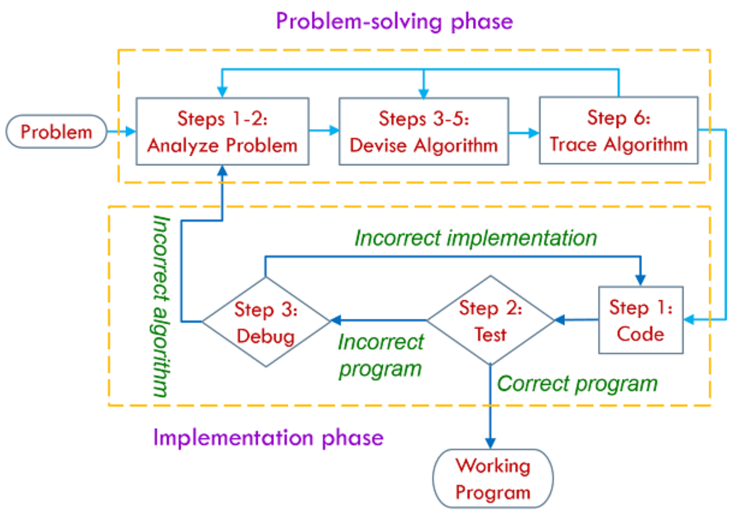
Program Development Process

Computers are used to solve problems. However, before a computer can solve a problem, it must be given instructions for how to solve the problem. A professional programmer usually doesn't just sit down at a computer keyboard and start typing out. Instead, program development is a multi step process that requires the programmer to *understand the problem*, *develop an algorithm*, *write the program*, and then *test the program*. When you - the programmer - is given the task of developing a program, you'll be given a *program requirements statement* that specifies what the proposed program is to accomplish and the design of any program interfaces (that is, what sort of input the program would consume and what sort of output the program would produce). You would also receive an overview of the complete project so that you'll understand how your part fits into the whole. Your job then is to determine how to take the inputs you're given and convert them into the specified outputs. This is known as *program design* and sometimes colloquially referred to as *coding* and sometimes as *programming*.

To help structure program development, the program design process can be divided into two general phases: the *problem-solving* phase and the *implementation* phase, each of which is further subdivided into multiple steps. A high-level view of the program design process is illustrated in the following figure.



The problem-solving phase devises an algorithm that expresses a solution to the problem. This is the most critical component of the program design process because it provides a solution to the problem independent of any programming language. This phase is divided into six steps that collectively analyze the problem, work the problem by hand for selected input(s), find pattern(s) from the hand-computations to devise a step-by-step solution, and perform tests to ensure that the solution is correct.

The implementation phase uses a specific programming language to convert the algorithm received from the problem-solving phase into a working computer program. This phase comprises of three steps: coding, testing, and debugging.

Problem-Solving Phase: Distance problem

The six steps of the problem-solving phase are:

1. Understand the problem clearly.
2. Describe the input and output information.
3. Work the problem by hand for a simple data set.
4. Decompose solution from previous step into step-by-step details.
5. Generalize the steps into an algorithm.
6. Test the algorithm with a broader variety of data.

Step 1: Problem statement

The first step to solving any problem is to understand it. Begin by stating the problem clearly. It is extremely important to give a clear, concise problem statement to avoid any misunderstandings. To give you an idea of how this process works, a fairly simple problem from geometry and trigonometry is to find the *distance between two points*. Are you supposed to compute the distance for two points on a number line [one-dimensional], or on a plane [two-dimensional], or in space [three-dimensions], or in an abstract $n-$dimensional space? Once the problem is exactly specified [in your case, the problem is to compute the distance between two-dimensional points], begin by writing a concise problem statement:

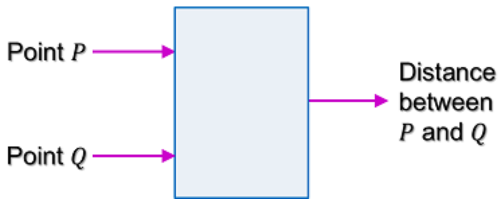
$$\text{Compute straight line distance between two points in a plane.}$$

As this example shows, even the simplest problem requires clarification. Imagine, how many questions you'd have for a problem statement consisting of hundreds or even thousands of detailed statements.

Step 2: Input/Output description

The second step is to carefully describe the information that is given to solve the problem and then identify the values to be computed. These items represent the *input* and the *output* for the problem and collectively can be called *input/output* (I/O).

For many problems, a diagram that shows the input and output is useful. At this point both the solution and the program are an abstraction because you're not defining the steps to determine the output; instead, you're only showing the information that is used to compute the output. The I/O diagram for the distance between two points P and Q follows:



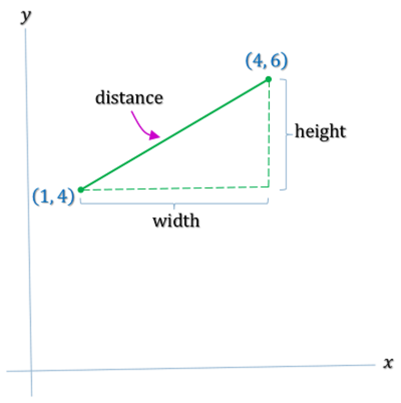
Step 3: Hand example

The third step is to work the problem by hand or with a calculator, using a simple set of data. This is a very important step, and should not be skipped even for simple problems. This is the step in which you work out the details of the problem solution. Often this step will involve drawing a diagram of the problem at hand, in order to work it precisely. The more precisely you can perform this problem [including the more precisely you can draw a diagram of the situation if applicable], the easier the remainder of your steps will be. A good example of the sort of picture you might draw would be the diagrams drawn in many science classes [especially physics classes].

To compute the distance between two points in a plane, the solution by hand would begin with points P and Q having the following coordinates:

$$P = (1, 4)$$
$$Q = (4, 6)$$

Based on your knowledge of analytic geometry, you would first draw a right triangle connecting points P and Q, as shown in the following figure, and then specify the distance between the two points as the right triangle's hypotenuse.



Using the Pythagorean theorem, you would then compute the distance with the following equation:

$$\begin{aligned} d(P, Q) &= \sqrt{(\text{width})^2 + (\text{height})^2} \\ &= \sqrt{(4-1)^2 + (6-4)^2} \\ &= \sqrt{13} \\ &= 3.605551 \end{aligned}$$

If you get stuck at this step, it typically means one of two things. The first case is that the problem is *ill-specified* - it is not clear from the problem statement what you are supposed to do. If you cannot take a simple set of numbers and compute the output [either by hand or with a calculator], then you're no where close to implementing a program to solve the problem. In such a situation, you must resolve how the problem should be solved before proceeding. In the case of a classroom setting, this resolution may require asking your professor or TA for more details. In a business setting, asking your technical lead or customer may be required. If you're solving a problem of your own creation, you may need to think harder about what the right answers should be and redefine your definition of the problem.

The second case where Step 3 is difficult is when you lack *domain knowledge* - the knowledge of the particular field or discipline the problem deals with. If you don't have knowledge of analytic geometry for the distance example, that would be an example of lacking domain knowledge - the problem domain is mathematics, and you are lacking in math knowledge. No amount of programming expertise nor effort [that is, "working harder"] will overcome this lack of domain knowledge. Instead, you must consult a source with domain expertise - a math textbook, website, or an expert. Once you have the correct domain knowledge, you can proceed with solving your instance of the problem. Note that domain knowledge may come from domains other than math. It can come from any field, as programming is useful for processing any sort of information.

Step 4: Writing precise instructions for solving particular instance

For this step, you must think about what you did to solve the problem in step 3, and write down the individual steps to solve that particular instance. That is, you must write down a clear step-by-step outline of the solution that could be followed by anybody else to reproduce your answer for the particular problem instance solved in Step 3. If you do multiple instances in Step 3, you'll repeat Step 4 multiple times as well, once for each instance you did in Step 3. If an instruction is somewhat complex, that is all right, as long as the instruction has a clear meaning - these complex steps will be solved separately later on by turning them into their own programming problems.

The difficult part of Step 4 is thinking about exactly what you did to accomplish the problem. The difficulty here is that it is very easy to mentally gloss over small details, "easy" steps, or things that you do implicitly. Implicit assumptions about what to do, or relying on common sense lead to imprecise or omitted steps. The computer will not fill in any steps you omit, thus you must be careful to think through all the details. This difficulty is best illustrated by the exercise of making a peanut butter and jelly sandwich. Here's my step-by-step outline for preparing such a sandwich:

1. Grab some bread
2. Toast the bread
3. Get some peanut butter and jelly
4. Spread the peanut butter on the bread
5. Spread the jelly on top of the peanut butter

Most humans would be able to follow these instructions to make a sandwich since they have the ability to understand the context, the ability to "read between the lines", and based on the context or situation determine the meaning of what was said. This is a skill that computers lack. A computer would not be able to parse meaning from "Grab some bread". If computers could speak, the litany of questions asked by a computer would include: What is "bread"? What does "grab" mean? "Grab" from where? What is "toast"? How to "toast"? When to know bread is toasted? What is "peanut butter"? And, "jelly"? Where are they stored? How to "Get some peanut butter and jelly?" What does "Spread" mean and so on. Since computers are completely literal - they do exactly what they're told and nothing more - the step-by-step outline must be precise, specific, and thorough.

Returning to the distance problem, the exact steps for computing the distance between two points in a plane would look like this:

Input: $P = (1, 4)$ and $Q = (4, 6)$
Output: $d(P, Q) = 3.605551$

1. Compute $width$ of right triangle :
  Subtract $P.x$ from $Q.x$
  You get $4 - 1 = 3$
2. Compute $height$ of right triangle :
  Subtract $P.y$ from $Q.y$
  You get $6 - 4 = 2$
3. Compute $(width)^2$:
  Multiply 3 by 3
  You get 9
4. Compute $(height)^2$:
  Multiply 2 by 2
  You get 4
5. Compute $(width)^2 + (height)^2$
  Add 9 and 4
  You get 13
6. Compute $\sqrt{(width)^2 + (height)^2}$
  Apply square root operation on 13
  You get 3.605551

The steps are very precise leaving nothing to guess work. Anyone who can perform basic arithmetic can follow these steps to get the right answer. Computers are very good at arithmetic, so none of these steps is even complex enough to require splitting into a subproblem.

Step 5: Generalize specific steps into algorithm

Once you've solved one or more instances of the problem, you're ready to generalize the steps into an algorithm. You've to use the insights gained from solving specific instances to find a pattern that allows you to solve the problem for possibly infinite instances. This generalization typically requires two activities. First, you must take particular values that you used and replace them with mathematical expressions of the input parameters. The second activity to generalize steps requires you to find repetition of certain patterns - that is, find particular steps that are repeated over and over. You must generalize how many times to repeat the pattern, as well as what the individual steps of the pattern are. In the current distance example, repetition is unnecessary - a second example is shown later on to illustrate how finding repetition of certain steps will help devise an algorithm.

For simple problems such as the distance problem, the algorithm can be listed as operations that are performed one after another. Replacing the particular values used with mathematical expressions of the input parameters, steps 3, 4, 5, and 6 are combined into a single step.

Algorithm $d(P, Q)$

Input: $P = (P.x, P.y)$ and $Q = (Q.x, Q.y)$
Output: distance $d(P, Q)$ between points P and Q

1. $width = (Q.x - P.x)$ [Compute width of right triangle generated by points P and Q]
2. $height = (Q.y - P.y)$ [Compute height of right triangle generated by points P and Q]
3. $d(P, Q) = \sqrt{(width)^2 + (height)^2}$ [Compute hypotenuse of right triangle]

Step 6: Tracing the algorithm

The final step of the problem-solving phase is to test the algorithm devised to solve the problem. The algorithm must first be tested with the hand examples from Step 3 because those results are already computed. Once the algorithm works for the hand-calculated examples, it should also be tested with additional sets of data to be sure that the algorithm works for other valid data sets.

$\textbf{Algorithm}\ d(P, Q)$ receives the locations of two points P and Q in a plane and produces as output, $d(P, Q)$, the distance between the two points. You show how the algorithm executes if the input points have locations $P(1, 4)$ and $Q(4, 6)$. Simulating the execution of an algorithm for a specific input is called a *trace*. At line 1, the algorithm computes $width$ - the width of the right triangle generated by points P and Q. At line 2, the algorithm computes $height$ - the height of the right triangle generated by points P and Q. The effect is to assign values 3 and 2 to $width$ and $height$, respectively. At line 3, the algorithm computes the length of the hypotenuse of the triangle, which is equal to the distance, $d(P, Q)$, between points P and Q. At this point, the algorithm updates $d(P, Q)$ with the value $\sqrt{13}$ which has an approximation of 3.605551.

Although the two-dimensional plane consists of four quadrants, the previous test of the algorithm uses points located in the first quadrant. The trace of the algorithm for the second quadrant begins by choosing a pair of second quadrant points $P(-1, 4)$ and $Q(-4, 6)$. At line 1, the algorithm computes $width$ as -3. This value should be problematic since it is geometrically impossible for the length of triangle's edge to be negative. That is, until you notice that in line 3, the right triangle's hypotenuse is computed using the square of $width$ - which is always a positive value. At line 2, the algorithm computes $length$ as 2. The resultant distance $d(P, Q)$ in line 3 is updated with the expected approximation 3.605551.

The algorithm is traced for points in the third-quadrant by setting values $P(-1, -4)$ and $Q(-4, -6)$ and for points in the fourth-quadrant by setting values $P(1, -4)$ and $Q(4, -6)$. In both cases, the distance $d(P, Q)$ in line 3 is updated with the expected value 3.605551.

The distance between two points located at the same position must be 0. Tracing the algorithm for two coincident points leads to lines 1 and 2 assigning 0 to both $width$ and $height$, and subsequently, line 3 evaluating this as $\sqrt{0}$ which is equivalent to 0.

It is well known that the square root of a negative number cannot be a real number. Mathematicians have developed a system of imaginary numbers to express square roots of negative numbers. However, computers are only able to represent a range of integral numbers and a subset of real numbers. And, hence it is important for programmers to test whether the algorithm is being asked to execute a step requiring the computation of the square root of a negative number. In your case, line 3 computes the square root of a value that is obtained by adding two positive numbers [why?] and therefore will never compute the square root of a negative number.

Implementation Phase: Distance problem

The three steps of the implementation phase consist of:

1. Coding the algorithm.
2. Testing the code.
3. Debugging the code if step 2 doesn't match output generated by hand-calculations.

Step 1: Coding

After an algorithm has been devised in Step 5 of the problem-solving phase that solves the original problem and after it has been thoroughly hand-tested in Step 6 of the problem-solving phase, the solution can be coded for execution by a computer. If the algorithm has been given in sufficient detail, much of the coding will be routine. However, every programming language has its own peculiarities, so additional issues will have to be dealt with.

$\textbf{Algorithm } d(P, Q)$ is converted to a C program so that you can use a computer to perform the computations:

```c
/*
Source file: distance.c

This program prompts user to enter source and destination points P and Q;
computes the distance between points P and Q using Algorithm d(P,Q);
and prints the computed distance to standard output.

To compile using GCC on Linux:
gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror distance.c -o dist-gcc.out -lm
To compile using clang on Linux:
clang -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -lm distance.c -o dist-clang.out -lm
To compile using Microsoft C compiler (in Windows):
cl /EHsc /TP /Za /Fedist-msvc.exe distance.c
*/

#include <stdio.h>
#include <math.h>

int main(void) {
  // input portion
  double px, py, qx, qy;
  printf("Enter point P: ");
  scanf("%lf %lf", &px, &py);
  printf("Now, enter point Q: ");
  scanf("%lf %lf", &qx, &qy);

  // computations
  double width = qx - px;
  double height = qy - py;
  double distance = sqrt(width*width + height*height);

  // output portion
  printf("Distance from P(%.3f, %.3f) to Q(%.3f, %.3f) is %.3f\n",
         px, py, qx, qy, distance);
  return 0;
}

```

The source code in `distance.c` can be compiled into an executable machine code like this:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror
distance.c -o dist-gcc.out
```

Note that the `$` character above represents the Linux bash shell prompt and is not a part of the compiler command.

In Linux, the executable generated by the compiler can be run like this:

```
$ ./dist-gcc.out
Enter point P: 1 4
Now, enter point Q: 4 6
Distance from P(1.000, 4.000) to Q(4.000, 6.000) is 3.606
$
```

Step 2: Testing

After the solution has been coded, testing proceeds similarly to hand-testing an algorithm. The difference, of course, is that the code will now be executed by the computer several times for different input. Again you should try to make the input as difficult and potentially troublesome for the computer as possible. Boundary values and input for which the solution should be trivial should again be considered. Large input sizes, infeasible to hand-test, must now be tested.

Program testing can be a very tedious and time-consuming part of program development. This is surely one major factor for students submitting code [for assignments] that works for a particular input but not for other inputs. As a programmer, you're responsible for completely testing your program. In business environments consisting of large development projects, there are often specialists known as test engineers who're responsible for testing to make sure the programs written by individual programmers and teams work together as a single system.

There are two types of testing: *blackbox* and *whitebox*. Blackbox testing is done by the system test engineer and the user. Whitebox testing is the responsibility of the programmer.

Blackbox testing gets its name from the concept of testing the program without knowing what is inside it, that is, without knowing how it was designed and implemented. In other words, the program is like a black box that cannot be seen into. Instead, the test engineer uses the problem statement to determine the program's behavior, then supplies input data data to be consumed by the program, and then compares the output data produced by the program with the expected output obtained through hand-testing.

Whereas blackbox testing assumes that the tester knows nothing about the program, whitebox testing assumes that the tester knows everything about the program. In other words, the program is like a glass house in which everything is visible. For this reason, whitebox testing is also referred to as *glass house testing*. Whitebox testing is your responsibility. As the programmer, you know exactly what is going on inside every program you author. You must make sure that every command, every instruction, and every possible situation have been tested. This is not a simple task!

An important aspect of whitebox testing is to test functions of a program (*function* is the term used in C to describe code that encapsulates an algorithm) in isolation. You write a short program, called a *test harness*, that calls the function to be tested and verifies that the results are correct. This process is called *unit testing*. You must begin thinking of unit testing when you're designing the algorithm. Ask yourself what boundary conditions and unusual situations you need to test for and make a note of them immediately because being human you won't remember them an hour later. When you're writing your pseudocode or drawing your flowcharts, review them with an eye towards test cases and make additional notes of the cases you need. Always document your test cases. After your program is finished and in production, you'll need the test plan again when you make modifications to the program. Finally, remember that except for the most trivial programs, one set of test data will not completely validate a program. Large-scale development projects may require hundreds of test cases to validate a program.

What happens when programs malfunction? Catastrophic failures of systems attributed to incorrect software behavior are cataloged here and here. This article discusses what happens when software executes incorrect and unexpected actions.

How do you know when a program is completely tested and 100% reliable? In reality, there is no way to know for sure. But you can do certain things to keep the odds on your side. Even though not all of the concepts may be clear to you until more material and topics are covered in this course, they're included here for the sake of completeness:

1. Verify that every line of code has been executed at least once.
2. Verify that every selection statement in your program has executed both $true$ and $false$ branches.
3. For every iteration statement in your program, make sure the tests include the first and last iterations. The tests must also include iterations below the first and above the last to ensure that out-of-range iterations are not executed by the program.
4. If error conditions are being checked, make sure all error logic tested. This may require you to make temporary modifications to your program to force the errors. For instance, an input/output error usually cannot be created - it must be simulated.

Step 3: Debugging

After a program is completely tested and you've ensured that the program produces correct output for specified input, you have now created a *working program* that can then be used by your clients. Instead, if the program generates incorrect output or behaves in unintended ways, then you've created an incorrect program containing one or more software bugs. Bugs can arise because of *incorrect implementation* - that is, mistakes and errors were made in the coding step [Step 1 of the implementation phase]. Bugs can also arise because of incorrect design - that is, an *incorrect algorithm* was conceived during Step 5 of the problem-solving phase.

The intent of the debugging step is to identify the source of software bugs within the program. The activity of debugging is distinct from testing [Step 2 of the implementation phase]. While testing focuses on revealing the presence of bugs [by finding inputs for which the program generates incorrect outputs], debugging is concerned with the processes initiated after incorrect program behavior is detected by testing; hence, debugging follows testing.

Once testing has determined the existence of bugs, you begin the process of locating the source of the bug and fixing it. The most popular way to find bugs is by *manual debugging*; that is, you manually contrast what is happening to what should be happening when your program is executing. The intent here is to confirm, one by one, that the many things you believe to be true about your program are actually true. The classic technique is to insert *trace code* into the program to print values of variables as the program executes. By manually analyzing the output of the trace code, you may find that one or more of these variables have values that contrast with the correct values they must have for correct program behavior. In that case, you've found a clue to the location of a bug. Further analysis of the trace code output and hand simulation of the program will help you determine whether the bug was caused by incorrect implementation or by an incorrect algorithm. As illustrated by Figure 1, bugs caused by incorrect implementation are easier to correct and only require an iteration over the implementation phase. Instead, bugs caused by incorrect algorithm will be more expensive since they will require a re-evaluation of the entire program design process.

What makes debugging time consuming and expensive is that it is an iterative process. Debugging requires programmers to implement a constant cycle of adding trace code to print out values of variables during execution, recompiling, executing the program, analyzing the output of the trace code, and then moving the trace code to a different location after removal of the current bug.

To speed up the debugging process, modern compiler environments provide special programs, called *debuggers*, that remove the tedium of inserting trace code into your program. A debugger is a special program that executes other programs such as your buggy program, allowing you to run your program step by step by providing facilities to pause and restart the program's execution and at each step examine the values of variables. A debugger allows you to be more productive by directly examining what your program is doing while it runs without the need for trace code.

I'll provide more details as the semester unfolds on debugger usage and special strategies for debugging that will make you more productive as a programmer.

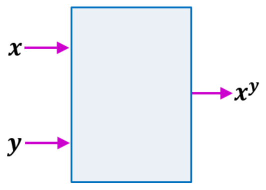
Problem-Solving Phase: Exponentiation

Consider the problem of writing an algorithm for computing the exponentiation function x^y.

The first step is to understand the problem clearly. One understanding of the exponentiation function x^y is by substituting suitable values for x and y. In mathematics, the set of real numbers contains the set of infinite points on the infinitely long real number line ranging from $[-\infty, \infty]$. Exponentiation is used extensively in many fields, including economics and physics, chemistry, and biology with applications such as population growth, compound interest, and chemical reaction kinetics. For our solution, we restrict the values of x and y to integer values (because the exponentiation algorithm is simpler for integer values) so that we can think of $x^y = \underbrace{x \times \cdots \times x}_{y \text{ times}}$.

Understand that if $x = 2$ and $y = 3$, then $2^3 = 8$. Further, confirm your understanding with another set of values, say $x = 3$ and $y = 4$, and compute $3^4 = 81$.

Because of our clear understanding of the problem, the second step of describing inputs and output becomes straightforward: both inputs x and y are integer values and the output x^y is also an integer value. For now, the algorithm is unknown and can be thought of as a blackbox allowing us to visualize the inputs to the blackbox and output generated by the blackbox as:



The third step is to work the problem by hand or with a calculator, using a simple set of data. Suppose $x = 3$ and $y = 4$. The steps to compute x^y or 3^4 by hand will look like this:

$$\begin{aligned} &\text{Input:} \quad x = 3 \text{ and } \; y = 4 \\ &\text{Output:} \; x^y = 3 \times 3 \times 3 \times 3 \times = 81 \end{aligned}$$

For the 4^{th} step, you must think about what you did to solve the problem in step 3, and write down the individual steps to solve that particular instance. That is, you must write down a clear step-by-step outline of the solution that could be followed by anybody else to reproduce your answer for the particular problem instance solved in Step 3. Step 4 for the hand solution in step 3 would look like this:

$$
\begin{array}{l}
\text{Input:}\quad x = 3 \text{ and } \; y = 4 \\
\text{Output: } x^y = 81 \\
\\
\textbf{1. } \text{Multiply } 3 \text{ by } 3: \\
\qquad \text{You get } 9 \\
\textbf{2. } \text{Multiply } 3 \text{ by } 9: \\
\qquad \text{You get } 27 \\
\textbf{3. } \text{Multiply } 3 \text{ by } 27: \\
\qquad \text{You get } 81 \\
\textbf{4. } 3^4 = 81
\end{array}
$$

For the 5^{th} step, you've to use the insights gained from step 4 to find a pattern that allows you to solve the problem of computing x^y for not just $x = 3$ and $y = 4$ but for possibly infinite combinations of integer values x and y. The first task in step 5 is to replace particular values used in each step of step 4 with mathematical expressions of parameters. Doing this for step 4 would look like this:

$$
\begin{array}{l}
\text{Input:}\quad x = 3 \text{ and } \; y = 4 \\
\text{Output: } x^y = 81 \\
\\
\textbf{1. } \text{Set variable } n = x \\
\textbf{2. } \text{Multiply } x \text{ by } n = 3: \\
\qquad \text{You get } n = 9 \\
\textbf{3. } \text{Multiply } x \text{ by } n = 9: \\
\qquad \text{You get } n = 27 \\
\textbf{4. } \text{Multiply } x \text{ by } n = 27: \\
\qquad \text{You get } n = 81 \\
\textbf{5. } x^y \text{ is } n
\end{array}
$$

The second task in step 5 is to find repetition in terms of these parameters:

$$
\begin{array}{l}
\text{Input:}\quad x = 3 \text{ and } \; y = 4 \\
\text{Output: } x^y = 81 \\
\\
\textbf{1. } \text{Set variable } n = x \\
\textbf{2. } n = \text{Multiply } x \text{ by } n \\
\textbf{3. } n = \text{Multiply } x \text{ by } n \\
\textbf{4. } n = \text{Multiply } x \text{ by } n \\
\textbf{5. } n = x^y \text{ is result}
\end{array}
$$

Notice that after setting $n = x$, the repetitive process of multiplying x by n occurs 3 times where 3 is equivalent to $y - 1$. Using this key insight, the third and final task in step 5 is to write the algorithm as:

```
Algorithm $x^y$
Input:  Integer values: $x$ and $y$
Output: $x^y = \underbrace{x \times \cdots \times x}_{y \text{ times}}$
1. $n = x$
2. $i = 1$
3: while $(i < y)$
4:   $n = n \times x$
5:   $i = i + 1$
6: endwhile
7. print value of $n$
```

Implementation Phase: Exponentiation

Algorithm x^y is converted to a C program so that you can use a computer to perform the computations:

```c
/*
Source file: expo.c

This program prompts user to enter two integer values, say x and y.
The program computes and prints x^y to standard ouput.

To compile using GCC on Linux:
gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror expo.c -o expo-gcc.out -lm
To compile using clang on Linux:
clang -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -lm expo.c -o expo-clang.out -lm
To compile using Microsoft C compiler (in Windows):
cl /EHsc /TP /Za /Feexpo-msvc.exe expo.c
*/

#include <stdio.h>

int main(void) {
  // input portion
  printf("Enter integer values for base and power: ");
  int base, power;
  scanf("%d %d", &base, &power);

  // computations
  int expo = base;
  int i = 1;
  while (i < power) {
    expo = expo * base;
    i = i + 1;
  }

  // output portion
  printf("pow(%d, %d) = %d\n", base, power, expo);
  return 0;
}

```

The source code in `expo.c` can be compiled into an executable machine code like this:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror
expo.c -o expo.out
```

In Linux, the executable generated by the compiler can be run like this:

```
$ ./expo.out
Enter integer values for base and power: 3 5
pow(3, 5) = 243
$
```