Introduction to Algorithms and Pseudocode

Nicholas Bennett
nickbenn@g-r-c.com

August 2015

Copyright and License

Last modified: 27 August 2015.

Algorithms

What is an Algorithm?

An algorithm is a finite, explicit step-by-step procedure for solving a specific problem or accomplishing a specific goal. We frequently talk about algorithms in mathematical terms, and many algorithms are expressed using notation borrowed from mathematics, but algorithms aren't necessarily mathematical in the operations performed, or in the results produced.

In general, an effective algorithm has these characteristics:

- Explicit, complete, and precise initial conditions;
- A finite, complete, unbroken – but not necessarily purely linear – series of steps to follow, to arrive at the desired result;
- Explicit, complete, and precise terminal (stopping) conditions. If appropriate, these should include conditions that indicate that the problem can't be solved by the algorithm.

The steps in an algorithm should be sufficient to go from the initial conditions to the intended goal, or to a condition in which it's clear that the algorithm won't produce the desired result. This latter outcome doesn't necessarily mean that it's a bad algorithm. Similarly, when an algorithm doesn't solve a problem, that doesn't necessarily mean the problem can't be solved – only that that specific instance and formulation of the problem can't be solved with that algorithm.

To be effective, it's important that an algorithm is unambiguous, at least in its critical elements. For example, we probably wouldn't place much confidence in an algorithm that included the instruction: "Step 3: Subtract 18, or maybe 43, from the total."

How are Mathematical Statements and Algorithms Related?

Generally, a mathematical statement isn't an algorithm – but such statements are often key elements of algorithms. A statement uses constants, variables, operators, and functions to describe a relationship between mathematical entities – but it doesn't tell us (at least not directly) what to do with that relationship.

Example: Converting Fahrenheit to Celsius

We can describe the relationship between the Celsius and Fahrenheit scales as follows:

$$5(F-32) = 9\mathrm{C} \tag{1}$$

where

F = temperature in degrees Fahrenheit, and
C = temperature in degrees Celsius.

Formula (1) defines the relationship between temperatures in Celsius and Fahrenheit, but it doesn't give us an explicit algorithm for converting from one to the other. Fortunately, if we have some understanding of algebra, we can easily write such an algorithm.

First, we can use the basic operations of algebra to convert (1) into a form that expresses *C* as a function of *F*:

$$C = \frac{5(F-32)}{9}. \tag{2}$$

Now it's a straightforward task to write an algorithm (based on the standard order of arithmetic operations) to convert from Fahrenheit to Celsius.

1. Start with a given temperature in degrees Fahrenheit.
2. Subtract 32 from the value used in step #1.
3. Multiply the result of step #2 by 5.
4. Divide the result of step #3 by 9.
5. The result of step #4 is the temperature in degrees Celsius.

Algorithm 1: Conversion from Fahrenheit to Celsius

Note that we've shifted from describing the relationship between the two temperature scales to specifying a sequence of computational steps. That's a key difference between a mathematical statement and an algorithm.

What Do Algorithms Have to Do With Computers?

Computer programming consists, in large part, of creating unambiguous, step-by-step procedures for the computer to follow, to produce specific results. In other words, we might say that computer programming is almost all about algorithms.

In many respects, computers are electronic idiot savants: they can perform amazing feats of calculation and memory, but without our help they're almost totally incompetent when it comes to applying those abilities to practical problems. Want to compute the sine of an angle? How about computing the natural logarithm of a number? These tasks are easy for a computer – in fact, in most modern computers, these operations are built in to the CPU itself. But if you want to plot the graph of $y = \sin x$ on the screen, or balance your checkbook, or compute the area of the region $1 \le x \le 100$, $0 \le y \le 1/x$, the computer is helpless – until someone writes algorithms to accomplish these tasks, and "teaches" them to the computer.

Fortunately for us, many such algorithms have already been written, in languages the computer can understand. When we write a line of Python or Java code, when we compose a formula for the cell of an Excel spreadsheet, even when we fit a series of StarLogo TNG blocks together, we're using the algorithms others have written as building blocks for those we create.

Questions for Discussion

- What are some step-by-step procedures that you follow on a regular basis?
- Which of these are primarily quantitative, and which are more qualitative in nature?
- Do these algorithms have complete initial and terminal conditions?
- What completely unambiguous algorithms can you think of?
- What common algorithms can you think of that include some ambiguity? In those cases, does the ambiguity affect our ability to obtain the desired result?
- Is there another sequence, different from the one shown, in which we could perform the necessary steps to convert from Fahrenheit to Celsius? In other words, is there another algorithm for performing this conversion?
- In writing a temperature conversion algorithm, would you approach the problem differently if the target audience consisted of 7^{th} grade schoolchildren, vs. a group of engineers? If so, how would the two descriptions differ?
- Would you approach the problem of writing the algorithm differently if you knew that the target audience would be using calculators, vs. pencil and paper? What if they were using computers with Microsoft Excel installed?

Pseudocode

What is Pseudocode?

Sometimes, when we do a very careful job of articulating an algorithm, the result is precise, unambiguous, and clearly structured enough that a computer implementation of the algorithm can be written with little or no additional preparation – even by a programmer who has no prior familiarity with the algorithm in question. In this case, we might say that our description of the algorithm is actually *pseudocode*: it has many characteristics in common with – and may appear to be – programming language code, but it's not directly usable as such.

As a deliberate form of expression, pseudocode can be very useful for specifying the logic of a computer program (or some critical portion of a program) prior to that program being written. It's also useful for documenting the logic of a computer program after the fact. It can be used to express the high-level logic of an entire program, the lower-level details of a core function in an operating system or run-time library, the behaviors and methods in an agent-based or object-oriented program, and everything in between. But as useful as pseudocode can be, there's a catch: unlike actual programming languages, and unlike natural languages, there's no standard vocabulary or grammar for pseudocode. Pseudocode can be expressed in virtually any written language in existence. It can look very much like natural language – or it can appear so close to a programming language that at first glance we might think that's what it is.

So what is pseudocode? Simply put, it's a set of practices and conventions for producing very precise, minimally ambiguous descriptions of algorithms. One way this can be accomplished is through the use of algebraic conventions for variable naming and expressions, as well as specialized notation from set theory, linear algebra, and other branches of mathematics. The use of these mathematical conventions can go a long way toward reducing ambiguity in the description of an algorithm.

Ultimately, the most important characteristic of pseudocode is not really *what it is*, but *what it enables.* Working from well-written pseudocode, virtually any programmer with professional competence in a given programming language should be able to implement the algorithm described by the pseudocode, in the given language, with little or no need for further instruction.[1]

Writing Pseudocode

Given the fact that there isn't a pseudocode standard, we're mostly left to our own devices to come up with a suitable grammar and vocabulary for the pseudocode we write (unless, of course, we happen to be working in or for an organization which has established standards or conventions for pseudocode). But others have gone before us, and we can learn from them.

---

1 Of course, pseudocode isn't a silver bullet: bad pseudocode – like poorly written specifications of any kind – can certainly lead to bad code. Nor is the use of pseudocode a requirement for learning about, developing, or implementing algorithms. In fact, some leading computer scientists – including Robert Sedgewick – recommend against the use of pseudocode for describing algorithms in educational or reference materials, preferring instead that they be expressed in Java or other widely used programming languages.

1. Avoid mixing and matching natural languages (just as you should when naming variables and methods in program code). For example, if you're writing pseudocode in English, avoid including terms or variable names from other languages, unless there's a compelling reason to do so.
2. Strive for consistency – except when doing so would make the pseudocode less clear. For example, if in one part of your pseudocode you use a particular symbol to indicate assignment of a value to a variable, use that same symbol for all assignment operations.
3. Where a step in the algorithm is expressed primarily in natural language, with few symbolic notations, try to use proper grammar. However, remember that mathematical expressions are often less ambiguous than natural language, even with correct grammar.
4. Since most programming languages borrow keywords from English, pseudocode written in English usually resembles programming code to some extent. However, pseudocode shouldn't be tied to any single programming language; instead, it should employ concepts that are common to many of them. For example, most programming languages include *if-then-else*, *for-next*, and *while* flow control constructs, as well as function definition and invocation; combined with common symbols for calculating mathematical expressions and assigning values to variables, these are sufficient for expressing almost any algorithm.
5. It isn't necessary (or even desirable, in many cases) to have a one-to-one correspondence between each line of pseudocode and a corresponding line of program code, or between the symbolic names used in pseudocode and those used in program code. In particular, many common high-level operations (e.g. sorting values, searching for a minimum or maximum value from a list, reading a value from a file) should generally be stated in a single line of pseudocode, rather than including all of the steps necessary to perform those operations – unless, of course, the algorithm being described is for performing just such an operation.
6. Use indentation to make any non-linear structure apparent. For example, when using an *if-then* statement to show that some portion of the algorithm should be performed conditionally, place the conditionally executed portion immediately below the *if-then* statement, and indent it one tab stop to the right of the *if-then* statement itself. Similarly, if some portion of the algorithm is to be performed iteratively, under the control of a *for-next* or *while* statement, place that portion of the algorithm immediately below the *for-next* or *while* statement, and indent it one tab stop further to the right. (These are useful practices for program code as well.)
7. If an algorithm is so long or complex that the pseudocode becomes hard to follow, try breaking it up into smaller, cohesive sections, each with its own title; we can think of these sections as the pseudocode analogues to methods, functions, and procedures. When you do this, make sure that the pseudocode also includes an articulation of the higher-level sequence, showing the order in which the more detailed sections should be performed.

List 1: Pseudocode Guidelines

Examples

Sieve of Eratosthenes

Eratosthenes of Cyrene (c. 276 BCE – c. 194 BCE) was a mathematician and scientist, and a librarian at Alexandria [1]. He was generally considered to be an excellent all-round scholar, but not the leader in any individual field. Nonetheless, many of his accomplishments were impressive in their time (for example, he made surprisingly accurate estimates of the circumference and tilt of the Earth), and his most famous innovation, the *Sieve of Eratosthenes*, is still an important technique in number theory today.

A prime number is a positive integer which has exactly two *distinct* positive integral divisors: itself and 1. (By this definition, it's clear that 1 isn't a prime number.) We can prove that there's no limit to the number of primes, and no largest prime, but prime numbers slowly become more sparse as they increase in value. For example, there are 168 prime numbers between 1 and 1,000, but only 106 primes between 10,001 and 11,000, and only 81 between 100,001 and 101,000.

The Sieve of Eratosthenes is a simple and effective algorithm for identifying the primes in a range of numbers [2]. It's based on the fact that once we find a prime number, we've also found an infinite number of composite (non-prime) numbers – namely, all of the integral multiples of the prime that are greater than the prime itself. To find all of the primes between 2 and some upper limit, we simply remove all of the non-primes in that range, in a systematic fashion:

1. Write down all of the positive integers from 2 to the upper limit of the given range of numbers, in order.
2. Starting with the number 2, and proceeding in order to the largest integer less than or equal to the square root of the upper limit[2], do the following with each number:
    a. If the current number is not crossed-out:
        i. For all integral multiples of the current number, starting with its square, but not exceeding the upper limit:
            - Cross the multiple off the list.
3. Every number in the list that isn't crossed-out is prime.

Algorithm 2: Sieve of Eratosthenes, natural language

2 Any composite number can be expressed as the product of at least one pair of integer factors, both of which are greater than 1; one of the two factors in every such pair will always be less than or equal to the square root of the composite number. Thus, we need only eliminate the multiples of prime numbers less than or equal to the square root of the upper limit. Similarly, when we move to the next prime (i.e. the next lowest number that isn't crossed out yet) in step 2.a.i, the lowest multiple that we need to cross out is the square of that prime; lower multiples have already been crossed out.

The algorithm is described in fairly unambiguous terms, which makes for a good start. However, it probably wouldn't be considered pseudocode – not yet, anyway. For one thing, we know that pseudocode shouldn't be tied to a specific programming language – but neither should it be tied to a pencil-and-paper implementation, as our description of the Sieve of Eratosthenes is.

Let's make the algorithm more general, but also more formal. We'll use a few more mathematical expressions this time, but still nothing very advanced.

1. Let u = upper limit of the range of numbers in which we will look for primes.
2. Let $\boldsymbol{L}$ = ordered list of integers, initially containing the set of values $\{2, 3, 4, \ldots, u\}$.
3. Let $p = 2$ (the smallest value in $\boldsymbol{L}$).
4. While $p \leq \sqrt{u}$:
    a. Let $m = p^2$.
    b. While $m \leq u$:
        i. If m is in $\boldsymbol{L}$:
            - Remove m from $\boldsymbol{L}$.
        ii. Let $m = m + p$.
    c. Let p = the smallest value in $\boldsymbol{L}$ that is larger than the current value of p.
5. Done: The values remaining in $\boldsymbol{L}$ are the prime numbers between 2 and u, inclusive.

Algorithm 3: Sieve of Eratosthenes, mix of natural language and mathematical expressions

Note that the description of the algorithm no longer includes any details on the mechanics of setting up or updating the list of numbers (i.e. it no longer says things like “write down all of the positive integers …”, or “cross the multiple off the list”). But this change is good: it gives the developer flexibility in implementing those details, while still specifying the important aspects of the algorithm unambiguously.

Euclid's Algorithm

Euclid (*fl.* 300 BCE) was a prominent Greek mathematician – often called the "Father of Geometry" – who wrote and taught at the Library of Alexandria during the reign of Ptolemy I [3]. His most famous work, *Elements*, is arguably the most important mathematics textbook ever written.

While *Elements* deals primarily with geometry, the algorithm we'll explore now was a critical development in number theory, and it has many practical and theoretical applications even today.

Consider two line segments, of different lengths. Can we construct a third line segment, of such a length that this third line segment will measure the other two evenly (i.e. the third will fit into each of the other two an integral number of times, with no portion left over)? How can we find the largest such line segment?

Another way of expressing the problem is probably more familiar to you: Given two numbers, what's their greatest common divisor? However we state it, this is the problem solved by Euclid's algorithm [4].

The algorithm can be applied to many different kinds of numbers and algebraic quantities, including integers, rational numbers, and real numbers. However, the most common application is to positive integers; the pseudocode shown here assumes that's what we're dealing with.

In this algorithm, we'll use a few symbols you might not have seen before:

- $\in$ "Is in", "is an element of", or "is a member of". For example, $a \in \boldsymbol{B}$ indicates that a is a member of the set $\boldsymbol{B}$.
- $\mathbb{N}$ The set of natural or counting numbers; the set of positive integers: $\{1, 2, 3, \ldots\}$
- $\leftarrow$ "Gets", or "is assigned the value". For example, $a \leftarrow b$ means that the value of b is assigned to a (we could also state this as "let $a = b$", as we did in the pseudocode for the Sieve of Eratosthenes).[3] A less trivial example is $x \leftarrow (10y + z)$, which means that the value of y should be multiplied by 10 and added to the value of z, and the result assigned to x.

3 The most commonly used symbol for the assignment operator in pseudocode is =, which is also the assignment operator used in most programming languages. However, because = is also frequently used for equality testing in pseudocode (as well as some programming languages), this can sometimes lead to confusion. Another alternative to = and ← is :=, which is also used for assignment in programming languages derived from Pascal. In some cases, = and ← are both used in the description of an algorithm, with the former used for declaring or defining a symbol, and the latter denoting assignment of a value to a symbol.

With the symbols defined above, we can now write Euclid's algorithm in pseudocode:

- Given two numbers, a and b, where
  - $a \in \mathbb{N}$,
  - $b \in \mathbb{N}$.
- Define new variables a' and b', with
  - $a' \leftarrow a$,
  - $b' \leftarrow b$.
- While $(b' \neq 0)$:
  - If $(a' > b')$, then
    - $a' \leftarrow (a' - b')$;

    otherwise,
    - $b' \leftarrow (b' - a')$.
- a' is the GCD of a and b.

Algorithm 4: Euclid's algorithm

Note that as in most pseudocode, the indentation is significant. For example, the lines underneath and indented to the right of "While $(b' \neq 0)$" should be repeated as long as the condition $(b' \neq 0)$ is true.

Also, notice that this example doesn't use any numbering of the steps. Without such numbering, we'll assume that the algorithm proceeds from the first to the last step in order, except as modified by conditional or iterative execution. In this case, we can see that the third top-level step is an iterative *while* statement, and its dependent steps consist of an *if-then-else* statement. Thus, when we get to that step, we'll repeat the *if-then-else* statement until the condition for iteration with *while* is no longer true; then we'll move on to the fourth top-level step.

An experienced programmer (or mathematician) will probably recognize that the repeated subtractions in Euclid's algorithm can be expressed more concisely using the *modulo operation,*[4] with mathematically equivalent results [5]. (This change also makes the computation more efficient on most CPUs.) When writing or reading pseudocode, we shouldn't assume that code based on the pseudocode must follow it exactly, or that pseudocode written after the fact must follow the code exactly. Neither type of translation is a trivial or mechanical process, but one that demands the application of experience, thought, and judgment.

4 The *modulo operation*, written as a mod b, is the remainder produced when a is divided by b. When both a and b are positive, the result of the modulo operation is equal to the smallest non-negative value producible by zero or more subtractions of b from a.

Prim's Algorithm

In *graph theory*, a *graph* is a pair of sets – one set of *nodes* (points or vertices) and another of *edges*, where each edge connects exactly two of the nodes [6]. An *undirected graph* is one in which any edge can be traversed in either direction, and there is at most one edge between any pair of nodes. A *connected graph* is one in which a route can be found between any two nodes in the graph, by traversing one or more edges in the graph. A *weighted graph* is one in which each edge has a weight (which may be its length, cost of traversal, etc.).

In an undirected graph, a *tree* is a subset of the edges, connecting a subset of the nodes, so that there are no cycles – i.e. for any two nodes connected by edges in the tree set, there's only one path connecting those nodes that uses the edges in the tree set. A *spanning tree* is a tree which connects all of the nodes in an undirected graph; any connected graph contains at least one spanning tree. Finally, a *minimum spanning tree* (MST) is a spanning tree of a weighted, connected, undirected graph, which minimizes the total weight of the edges in the tree. In a given graph, there may be more than one spanning tree with the same minimum total weight.

The problem of finding the MST is an example of a combinatorial optimization problem. In such problems, the solution space consists of all the different possible combinations of decision variables; the solution task consists of finding the combination that satisfies the problem constraints, while minimizing or maximizing the value of some objective function. Many combinatorial optimization problems are very difficult to solve, since the number of possible solutions tends to grows much faster than the number of inputs. For example, the number of spanning trees in a *fully connected*, undirected graph (i.e. one in which there's a direct connection between every pair of nodes) with n nodes is n^{n-2}. A fully connected graph with 2 nodes has $2^0 = 1$ spanning tree; one with 5 nodes has $5^3 = 125$ spanning trees; one with 15 nodes has $15^{13} = 1{,}946{,}195{,}068{,}359{,}375$ spanning trees. Imagine using brute force to find the MST for a network with 1,000 nodes!

Fortunately, there are simple methods for finding the MST that are much more efficient than an exhaustive search. One of these is *Prim's algorithm*, named for Robert Prim, a mathematician and computer scientist who developed the algorithm in 1957 [7]. Actually, Prim independently reinvented an algorithm originally formulated in 1930 by Vojtech Jarník; because of this, the algorithm is sometimes called the Prim-Jarník algorithm [8]. (The algorithm was independently reinvented once again in 1959 by Edsger Dijkstra.)

Prim's algorithm is very straightforward, but the pseudocode version that follows uses some mathematical symbols that may be unfamiliar (though we used and described some of these symbols in the pseudocode for Euclid's Algorithm). The essential symbols are these:

- $\in$ "Is in," "is an element of," or "is a member of".
- $\notin$ "Is not in," "is not an element of," or "is not a member of". For example, $a \notin \boldsymbol{B}$ indicates that a is not a member of the set $\boldsymbol{B}$.
- $\emptyset$ The null, or empty set; a set with no members.

$\cup$ Union of sets. For example, $\boldsymbol{A} \cup \boldsymbol{B}$ is the set formed by the union of sets $\boldsymbol{A}$ and $\boldsymbol{B}$ – that is, the set of all elements that are contained in either one or both of $\boldsymbol{A}$ and $\boldsymbol{B}$ – but without duplicating any of the elements contained in both.

$\{\ldots\}$ The set containing the specified elements. For example, $\{a\}$ is a set containing the single element a, while $\{a, b\}$ is a set containing the elements a and b.

$\leftarrow$ "Gets," "takes the value," or "is assigned the value".

With the above definitions, we have this pseudocode for Prim's algorithm:

- Given the connected, weight graph $G(\boldsymbol{V}, \boldsymbol{E})$, where
  - $\boldsymbol{V}$ is the set of nodes;
  - $\boldsymbol{E}$ is the set of edges;
  - e_{uv} is the edge connecting the nodes u and v, where $u \in \boldsymbol{V}, v \in \boldsymbol{V}, e_{uv} \in \boldsymbol{E}$;
  - c_{uv} is the weight (cost, length, etc.) of e_{uv}.
- Define two additional sets, $\boldsymbol{E}_{mst}$ and $\boldsymbol{V}_{mst}$, with
  - $\boldsymbol{E}_{mst} \leftarrow \emptyset$ (set of edges in minimal spanning tree, initially empty),
  - $\boldsymbol{V}_{mst} \leftarrow \emptyset$ (set of nodes connected by minimal spanning tree, initially empty).
- Randomly select starting node $v_0 \in \boldsymbol{V}$.
- $\boldsymbol{V}_{mst} \leftarrow \{v_0\}$.
- While $\boldsymbol{V}_{mst} \neq \boldsymbol{V}$:
  - Find edge e_{uv} with minimum c_{uv}, where $u \notin \boldsymbol{V}_{mst}, v \in \boldsymbol{V}_{mst}, e_{uv} \in \boldsymbol{E}$.
  - $\boldsymbol{V}_{mst} \leftarrow \boldsymbol{V}_{mst} \cup \{u\}$.
  - $\boldsymbol{E}_{mst} \leftarrow (\boldsymbol{E}_{mst} \cup \{e_{uv}\})$.
- $\boldsymbol{E}_{mst}$ is a minimum spanning tree for $G(\boldsymbol{V}, \boldsymbol{E})$.

Algorithm 5: Prim's algorithm

Note that in the lines that start with "Randomly select …" and "Find edge …", it's assumed that we know how to select a random element from a set, and how to find the minimum-weight edge with one endpoint already connected to the MST, and one not yet connected. Operations such as these aren't unique to this algorithm, but are common operations used in many algorithms; thus, they're not described in detail here.

Monotone Chain Convex Hull Algorithm

In computational geometry, we sometimes need to find the *convex hull* of a set of points. Simply stated, the convex hull of a set of points is the minimal set that encloses the first set, and which is also *convex*.

First, what does *convex* mean in this context? You probably have a common sense understanding of the meaning, as applied to two- or three-dimensional shapes, but you might not know the actual definition. In fact, it's quite simple:

> A set of points is convex, *if and only if* for *any* pair of points in the set, all points on the line segment connecting those two points are also in the set [9].

As an example of how a mathematical statement expressed in natural language can also be expressed symbolically, we can also write the definition of a convex set as

$$\boldsymbol{S} \text{ is convex} \Leftrightarrow (\alpha\, \boldsymbol{p}_1 + (1-\alpha)\, \boldsymbol{p}_2) \in \boldsymbol{S},\ \forall\ \boldsymbol{p}_1, \boldsymbol{p}_2 \in \boldsymbol{S},\ \alpha \in [0, 1]. \tag{3}$$

If you find (3) confusing, try re-reading it, after reading these definitions:

- $\Leftrightarrow$ "If and only if". This is a relationship between two logical statements, and means that the two must either both be true, or both be false. For example, $a \Leftrightarrow b$ means that a and b are either both false, or both true.

- $\in$ "Is in," "is an element of," or "is a member of".

- $\forall$ "For all," or "for any". This is used to state that the statement to the left of the symbol applies for all conditions given to the right of the symbol, or that the operation described to the left of the symbol should be applied to all combinations described to its right. For example, $2\, x \in \mathbb{N},\ \forall\ x \in \mathbb{N}$ – i.e. for any value x that is a member of $\mathbb{N}$ (the set of natural numbers), the value $2x$ is also a member of $\mathbb{N}$.

- $[a, b]$ The set of real numbers bounded by the specified limits, including those limits. Another way to say this symbolically is to use the $\{\ldots\}$ set constructor notation – but instead of listing all of the values to be included in the set within the braces (which is impossible for real numbers), we would do it with $\{x \in \mathbb{R} \mid a \le x \le b\}$, which means the set of all real numbers x that satisfy the condition $a \le x \le b$.

Now that we have a solid understanding of (3), and thus what a convex set is, let's return to the task of finding the convex hull.

Imagine that each point in the first set of points is a nail driven part-way into a flat surface. Now, stretch an elastic band around the set of nails, so that all of the nails are within the perimeter of the band. Finally, let the elastic snap into place. The polygon formed by the elastic band is the boundary of the convex hull[5] of the points.

5 Some definitions refer to this boundary, rather than the set of points it encloses, as the convex hull.

At first glance, this might not seem like an interesting problem, but it has many applications in image processing, geographic information systems (GIS), statistics, and other fields. Also, while solving the problem might seem trivially easy (especially when we consider the rubber band analogy), remember that visualizing a solution in concept isn't the same as computing it.

Finding the convex hull is another combinatorial optimization problem. The task here is to find a subset of the specified points, and the permutation (ordering) of the points in that subset, so that they form the smallest possible convex polygon containing all the points. Fortunately, there are once again much better ways of solving the problem than making an exhaustive search through all possible subsets. One such method is the *monotone chain convex hull algorithm* (sometimes called *Andrew's monotone chain convex hull algorithm*, after its inventor, A. M. Andrew) [10], [11].

Andrew's algorithm is based on two intuitively obvious (and easily proved) characteristics of the convex hull, and one characteristic of all convex polygons:

- If the vertices of a convex polygon represent a subset of a set of points, and all of the remaining points in the set are either in the interior of the convex polygon or on the edges of the polygon, then that polygon is the boundary of the convex hull of the set of points.

- In the set of points, the two with the minimum and maximum X values are guaranteed to be vertices of the polygon forming the convex hull boundary. (This is also true of the points with the extreme Y values, but this algorithm doesn't take advantage of that fact.)

- If we were to start at one vertex of a convex polygon, and walk from vertex to vertex in a counter-clockwise direction, eventually returning to the starting point, we would turn to the left by some amount at every vertex. This also implies that if we encounter a right turn at some vertex, then the polygon is not convex.

Working from the above, the monotone chain convex hull algorithm consists of starting at the point with the minimum X value and working from left to right, provisionally adding each point to the lower portion of the polygon, after removing previously added points that don't result in turns to the left; when complete, this gives the lower portion of the convex polygon. The process is then repeated, starting at the point with the maximum X value and working from right to left; this gives the upper portion of the convex polygon.

For this algorithm description, we'll use the equals sign for the initial variable declarations, and ← to denote assignment of a value to an already defined variable. The only new symbols we'll use are these:

$|\boldsymbol{S}|$ The *cardinality* of $\boldsymbol{S}$; that is, if $\boldsymbol{S}$ is a set or a list with a finite number of elements, then $|\boldsymbol{S}|$ represents the number of elements in $\boldsymbol{S}$.

$\boldsymbol{K} \circ \boldsymbol{L}$ The concatenation of ordered lists $\boldsymbol{K}$ and $\boldsymbol{L}$. The result is a new list containing all the elements of $\boldsymbol{K}$ (in the same order), followed by all the elements of $\boldsymbol{L}$ (in order).

- Given
  - N = number of points.
  - $\boldsymbol{P}$ = set of points for which the convex hull is to be found, $|\boldsymbol{P}| = N$.
  - $\boldsymbol{P'}$ = working ordered list of points, initially empty.
  - $\boldsymbol{L}$ = ordered list of vertex points in lower portion of convex polygon, initially empty.
  - **U** = ordered list of vertex points in upper portion of convex polygon, initially empty.
  - $\boldsymbol{C}$ = final ordered list of vertex points in the convex polygon forming the boundary of the convex hull, initially empty.
- $\boldsymbol{P'} \leftarrow \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_N\}$, i.e. the points in $\boldsymbol{P}$, sorted in ascending order by X value, then by Y value (in case of tied X values).
- For $i \leftarrow 1$ to N:
  - While $|\boldsymbol{L}| \geq 2$, and the sequence of last 2 points in $\boldsymbol{L}$ and $\boldsymbol{p}_i$ do not make a left-hand turn:
    - Remove the last point from $\boldsymbol{L}$.
  - Append $\boldsymbol{p}_i$ to $\boldsymbol{L}$.
- For $i \leftarrow N$ down to 1:
  - While $|\boldsymbol{U}| \geq 2$, and the sequence of last 2 points in $\boldsymbol{U}$ and $\boldsymbol{p}_i$ do not make a left-hand turn:
    - Remove the last point from $\boldsymbol{U}$.
  - Append $\boldsymbol{p}_i$ to $\boldsymbol{U}$.
- Remove the last point from $\boldsymbol{L}$.
- Remove the last point from $\boldsymbol{U}$.
- $\boldsymbol{C} \leftarrow \boldsymbol{L} \circ \boldsymbol{U}$.

Algorithm 6: Monotone chain convex hull algorithm

Questions for Discussion and Activities

- After reading the examples, which form of expression (natural language, mathematical notation, or a mix of both) did you find easier to understand?
- Did your preference depend on the nature of the algorithm itself?
- Pick one of the predominantly natural language examples and translate it into a pseudocode expression that relies more on mathematical expressions. To see how well you did, present your results to a colleague. Did he or she understand the algorithm as you expressed it?
- Pick one of the predominantly mathematical expression-based examples and translate it into mostly natural language. To see how well you did, present your results to a colleague. Did he or she understand the algorithm as you expressed it?

Summary

In this document, we've introduced basic concepts of algorithms. We've also introduced practices for writing pseudocode, which can serve as an aid to the implementation of algorithms in computer programs; as a tool for documenting the algorithmic logic of existing programs; or simply as a formal, structured means of describing an algorithm, whether it is intended for implementation or not.

It's important to note that while an algorithm should have as little ambiguity as possible, that doesn't mean there's a single best way to express an algorithm, or a single best way to translate that expression into a computer program.

It would be a serious mistake to think that a software developer writing an implementation of an algorithm is little more than an automaton, performing a mechanical translation of the algorithm into code. Even in the few examples we've discussed here, there are a number of important details that are left unspecified, under the assumption that a skilled programmer will already know suitable methods for implementing those aspects of the algorithms, and specifying them is thus unnecessary. It's only natural that two equally competent developers, working from the same description of a non-trivial algorithm, using the same programming language, and following the same coding conventions, will often end up with different implementations. Even with a shared starting point, the effects of each programmer's unique background, skills, style, and creativity quickly become apparent.

The history, theory, and development of algorithms are rich areas of study, even when viewed apart from the world of electronic computation. But the ever-expanding role of scientific computing (*aka* computational science) makes an understanding of algorithms that much more important, and makes algorithms an even more rewarding subject of study and exploration.

References

[1] J. J. O'Connor and E. F. Robertson. “Eratosthenes of Cyrene.” Internet: http://www-history.mcs.st-and.ac.uk/Biographies/Eratosthenes.html, Jan. 1999 [Mar. 28, 2014].

[2] C. K. Caldwell. “The Prime Glossary: Sieve of Eratosthenes.” Internet: http://primes.utm.edu/glossary/xpage/SieveOfEratosthenes.html, 2014 [Mar. 28, 2014].

[3] C. Douglass. “Euclid.” Internet: http://www.mathopenref.com/euclid.html, 2009 [Mar. 28, 2014].

[4] Euclid (D.E. Joyce, ed.). (c. 300 BCE, ed. 1997.) *Elements* [Online]. Available: http://aleph0.clarku.edu/~djoyce/java/elements/toc.html [Mar. 28, 2014].

[5] “Modulo Operation.” Internet: http://en.wikipedia.org/wiki/Modulo_operation, Mar. 27, 2014 [Mar. 28, 2014].

[6] R. Diestel. *Graph Theory*, 3rd ed. Springer-Verlag, 2005, pp. 2-16.

[7] G. T. Heineman et al. *Algorithms in a Nutshell*. O'Reilly, 2008, pp. 169-171.

[8] J. L. Ganley. “Prim-Jarník algorithm.” Internet: http://xlinux.nist.gov/dads/HTML/primJarnik.html, Aug. 14, 2008 [Mar. 28, 2014].

[9] P. Sanchez et al. “Convex Set.” Internet: http://planetmath.org/encyclopedia/ConvexSet.html, May 31, 2010 [Mar. 28, 2014].

[10] A. M. Andrew. “Another Efficient Algorithm for Convex Hulls in Two Dimensions.” *Info. Proc. Letters*, vol. 9, pp. 216-219, Dec. 1979.

[11] “Monotone Chain Convex Hull.” Internet: http://www.algorithmist.com/index.php/Monotone_Chain_Convex_Hull, Nov. 11, 2011 [Mar. 28, 2014].