Introduction to Algorithms and Pseudocode

Nicholas Bennett
nick@nickbenn.com

January 2025

Copyright and license

Last modified: 27 January 2025.

Algorithms

What is an algorithm?

An algorithm is a step-by-step procedure for solving a specific problem or accomplishing a specific, finite goal. We frequently talk about algorithms in mathematical terms, but they're not necessarily mathematical in the operations performed, or in the results produced. On the other hand, many algorithms share not only some notation with mathematical formulæ, but also a certain level of formality in their expression.

In general, an effective algorithm has the following characteristics:

- Explicit, complete, and precise initial conditions.
- A complete, unbroken—but not necessarily purely linear—series of steps (operations, decision points, and jumps to other 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.
- All of the above expressed in symbolic terms, so that the algorithm can be applied to a any specific instance of a general class of problems.

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 the algorithm doesn't work; however, if the rules of the algorithm don't properly identify such a condition, then the algorithm is incomplete. Similarly, when an algorithm doesn't solve a problem to which it is applied, that doesn't necessarily mean the problem can't be solved—only that it 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 sometimes 43, from the total."

How are mathematical formulas and algorithms related?

A mathematical formula isn't necessarily an algorithm—but formulas are often key elements of algorithms. A formula describes a relationship between mathematical entities—variables and constants—but it might not tell us *what to do* with that relationship (at least not directly).

For example, the following formula describes the relationship between temperatures in the Celsius and Fahrenheit scales:

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

where

F = temperature in degrees Fahrenheit
C = temperature in degrees Celsius

The formula 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 the previous formula into this form:

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

Working from this formula, it's a relatively straightforward task to write an algorithm 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 scales to listing computational steps in a specific order. Also, note that the process required to convert (1) into (2) is itself an algorithm—one that you learned in first-year algebra—for isolating a variable in a linear equation.

What do algorithms have to do with programming?

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. Do you 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, they might even be 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 bounded by $1 \leq x \leq 100$ and $0 \leq y \leq 1/x$, 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 snap a collection of Scratch blocks together, we're using the procedures others have written as building blocks for the algorithms 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 not?
- Do these algorithms have complete initial and terminal conditions?
- What algorithms can you think of that include no ambiguity?
- What common algorithms can you think of that include some ambiguity? Does this ambiguity (or flexibility) 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 7th grade schoolchildren, vs. a group of engineers? If so, what would you change?
- Would you approach the problem of writing the algorithm differently if we knew that the target audience would be using calculators, vs. pencil and paper? How about if they were using Microsoft Excel on computers?

Pseudocode

What is pseudocode?

It may be that when we've done a careful job of spelling out the details of an algorithm, the result is so precise, so unambiguous, and so clearly structured 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 expression of the algorithm is actually a kind of *pseudocode*: it has many characteristics in common with programming language code, and it may appear very much like such code, but it is not, in fact, directly usable as code in any single programming language.

Pseudocode is a very useful device for specifying the logic of a computer program (or some critical portion of a program) prior to that program actually being written, as well as for documenting the logic of a computer program after the fact. It can be used to express the high-level logic of a traditional 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 is, 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 standard (albeit very formal) prose; at the other end of the spectrum, it can appear so close to programming code that on first glance, we might think that's exactly what it is.

So what is pseudocode? One way to describe it is easy, but arguably not very useful: in practice, pseudocode is simply a very precise, minimally ambiguous articulation of an algorithm—but even more precise, and less ambiguous, than usual.

Another clue comes from the world of mathematics: pseudocode often employs algebraic variable naming and expressions, as well as notation from set theory, linear algebra, and other branches of mathematics. The use of these mathematical conventions can go a long way toward eliminating, or at least reducing, ambiguity in the description of an algorithm.

Ultimately, the most important characteristic of pseudocode is not really what it *is*, but what it *makes possible.* As noted above, when we start with well-written pseudocode, virtually any programmer with reasonable 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.

Writing pseudocode

Given the fact that there isn't a standard pseudocode language, 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 well-established standards or conventions for pseudocode). But others have gone before us, and we can learn from them.

Pseudocode guidelines

1. Avoid mixing and matching of 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, unless doing so would make the pseudocode less clear. For example, if in one part of your pseudocode you use a particular symbol or verb to denote assignment of a value to a variable, use that same symbol or verb throughout.
3. Where a step in the algorithm is expressed primarily in natural language, with few symbolic notations, 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, it's to be expected that pseudocode will resemble programming code to some extent. However, pseudocode should not be tightly coupled with any single programming language. Instead, it should employ control structures, verbs, and other keywords that are common to most programming languages. For example, most imperative programming languages have *if-then-else*, *for-next*, and *while* flow control statements; when combined with mathematical symbols for calculation of simple expressions and assignment of values to variables, these are sufficient for expressing virtually 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, opening a file to read a value from it) should generally be stated in a single line of pseudocode, rather than including all of the steps necessary to perform those operations in practice—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 conditional 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. (By the way, these are good indentation practices for program code as well—even required in some cases.)
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.

Algorithm and pseudocode examples

Fisher-Yates shuffle

The shuffling problem is easily stated: How can we rearrange a list of items so that the order is random and fair? What constitutes a fair shuffle is an important question, but one that's a bit out of scope here; for our immediate purposes, we'll answer it in broad terms: A fair shuffle is one in which each of the possible outcomes (i.e. the different arrangements of our list of items) is equally likely to be produced.

Fortunately, this isn't a difficult problem to solve—but it's also easy to solve it incorrectly [1], [2]. There are two widely used, equally effective approaches to shuffling in computer programs: sorting on a random value, and the Fisher-Yates shuffle [3], [4]. The first is often used to shuffle rows of a spreadsheet, or records in a database; however, it is less efficient than the Fisher-Yates shuffle, which is the one we'll explore here.

Here's the original expression of the Fisher-Yates shuffle (paraphrased slightly):

1. Write a list of N items, inclusive.
2. Generate (by some appropriate means) a uniformly distributed random integer[1] k between 1 and the number of items that are not yet crossed out from the list (inclusive).
3. Counting from the first value not yet crossed out, cross out the k^{th} item that was not previously crossed out, and write that item at the end of a second list (initially empty).
4. Repeat steps 2–3 until all the items in the first list have been crossed out.
5. The second list of now has a random shuffling of the original list.

Algorithm 1: Pencil-and-paper form of the Fisher-Yates shuffle

The algorithm is described in fairly unambiguous terms, which makes for a good start. It's also adaptable to use with lists of practically any length, by using different values of N. 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; similarly, it shouldn't be tied to a pencil-and-paper implementation, as the preceding description is. Also, while we have some steps that are iterated as long as some condition holds, that structure isn't reflected visually.

1. A uniformly distributed random integer is one sampled from a range of integers, where the sampling process is such that each possible value has the same likelihood of occuring.

Let's add some indentation and symbols, and incorporate the idea (introduced to this problem by Richard Durstenfeld in 1964) of shuffling *in-place*—that is, without creating a second list, but instead swapping the positions of values in the first list. Beyond that, we'll keep the language fairly natural.

1. Let L be a list consisting of N elements: $(v_0, v_1, \ldots, v_{n-1})$.
2. For each value of i, counting backwards from $(n-1)$ to 1, inclusive:
    a. Generate a random integer j, uniformly distributed from 0 to i, inclusive.
    b. Swap elements v_i and v_j of L. That is, the element previously at position i will move to position j, and the element previously at position j will move to position i.
3. L is now shuffled.

Algorithm 2: Structured and symbolic form of Fisher-Yates shuffle

Note that the indentation (along with the sub-list numbering scheme) indicates that steps 2a and 2b will be repeated for each value i, starting with $(n-1)$ and counting down to 1, inclusive; when the countdown is finished, the algorithm moves to step 3.

Which of the two forms did you find easiest to understand? Which do you think would be easiest to explain to someone else? Which would be the clearest to work from while implementing the algorithm in a programming language.

Sieve of Eratosthenes

Eratosthenes of Cyrene (c. 276 BCE–c. 194 BCE) was a mathematician and scientist, and a librarian at Alexandria [6]. 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 measurements 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 is not a prime number, since it has only one positive integral divisor: itself.) 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 [7]. 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. So, 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:
    - If the current number is not crossed-out:
        - 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 3: Pencil-and-paper form of the Sieve of Eratosthenes

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. Conversely, when we move to a new number, the lowest multiple that we need to cross out is the square of that number.

Once again, we have a pretty clear description, but not quite pseudocode. Let's make it a bit more general, and more formal (which should also leave less room for ambiguity). We'll use a few more mathematical expressions this time, but 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 numbers, 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 4: 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 a good thing: 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 [8]. His most famous work, *Elements*, is arguably the most important mathematics textbook ever written.

Elements deals primarily with geometry, but Euclid also addresses number theory in the book. In fact, this example was initially expressed in geometric terms by Euclid, but his solution is an important development in number theory.

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? In any event, this is the problem solved by Euclid's algorithm, used by Euclid in the proof of two propositions in Book VII of *Elements* [9].

The algorithm can be applied to many different kinds of numbers and algebraic quantities, including integers, rational numbers, real numbers, polynomials, etc. 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 more interesting 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 then 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 used for assignment in programming languages derived from Pascal (Pascal, Modula-2, Oberon, Delphi, etc.).

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 5: Euclid's algorithm

Note that the indentation is again significant. For example, the lines underneath and indented to the right of "While $(b' \neq 0)$" should be repeated until the condition $(b' \neq 0)$ is no longer 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 and logically equivalent results [10]. (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, judgment, and creativity.

4. Simply put, 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 resulting from 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 (i.e. points or vertices) and another of edges, where each edge connects exactly two of the nodes [11]. 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, or the cost of traversal).

In an undirected graph, a *tree* is a set of 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. (There may be more than one spanning tree with the same minimum total weight in a given graph.)

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 is 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 a few hundred nodes!

Fortunately, this is a problem where there are a few simple solution techniques that work much better than an exhaustive search of the solution space. One of these is called *Prim's algorithm*—named for Robert Prim, a mathematician and computer scientist who developed the algorithm in 1957 [12]. Actually, Prim independently reinvented an algorithm originally invented in 1930 by Vojtech Jarník; because of this, the algorithm is sometimes called the Jarník-Prim (or Prim-Jarník) algorithm [13]. 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 to many (though we used and described two 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 $\boldsymbol{B}$.

$\varnothing$ 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 either in $\boldsymbol{A}$ or $\boldsymbol{B}$, or in both (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", or "is assigned the value".

With the above definitions, Prim's algorithm can be expressed as:

- 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:
  - $\boldsymbol{E}_{mst} \leftarrow \varnothing$ (set of edges in minimal spanning tree, initially empty)
  - $\boldsymbol{V}_{mst} \leftarrow \varnothing$ (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 6: 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 used in many algorithms; thus, they're not described in detail in the pseudocode.

Gift wrapping 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 of points which contains 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 [14].

We can write this definition symbolically as follows:

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

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. (Note that causation should not necessarily be assumed from this relationship.)
- $\in$ "Is in", "is an element of", or "is a member of".
- $\forall$ "For all", or "for any". This is used to state that the logical statement to the left of the symbol applies for all conditions described 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, $2x \in \mathbb{N},\ \forall\ x \in \mathbb{N}$—that is, 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 denotes the set of all real numbers x, such that $a \le x \le b$.

Now that we have a solid understanding of (3), and thus an understanding of 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.

This might not seem like an interesting problem at first, 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 imagination and visual intuition aren't strengths of most computers.

This is another combinatorial optimization problem; this time, the task is to find the subset of the specified points, and the permutation (ordering) of the points in that subset, so that they form a convex polygon that encloses all of the points. Fortunately, there are once again much better ways of solving the problem than making an exhaustive search through subsets of the points. One of the most intuitive methods, which performs well when the number of points on the boundary is relatively small, is the *gift wrapping algorithm* (sometimes called a *Jarvis march*, after its inventor, R. A. Jarvis) [15], [16].

Here's a two-dimensional version of the algorithm (though it can be extended to higher dimensions). In reading this, it might be helpful to imagine yourself walking along the perimeter of the convex polygon as it's being constructed.

1. Given a set of points on the XY Cartesian coordinate plane, and an initially empty ordered list of vertices for the convex polygon forming the boundary of the convex hull:
2. Find the point with the minimum Y value. This point is the *initial point* on the boundary polygon of the convex hull; it also becomes the *current point* in the algorithm.
3. Add the *current point* to the ordered list of vertices.
4. Find the *candidate point* in the set (excluding the *current point* from consideration) which, if a line is drawn from the *current point* through the *candidate point*, results in all of the other points in the set falling either on the left-hand side of the line, or exactly on the line itself.
5. If additional points (other than the *current point* and the *candidate point*) fall on the line, and the *candidate point* isn't the *initial point*:
    - Select the point exactly on the line which is furthest from the *current point*, in the same direction as the *candidate point*, to be the new *candidate point*.
6. Move to the *candidate point*, making it the new *current point*.
7. If this new *candidate point* is not the *initial point*, return to step #3 and proceed as before.
8. Done: The list of vertices, and the line segments connecting them (in the order in which the vertices were added to the list) make up the completed boundary of the convex hull.

Algorithm 7: Gift wrapping algorithm

Questions for discussion

- After reading the examples, which form of expression (i.e. using natural language or using mostly mathematical notation) did you find easier to understand?

- Did your preference depend on the nature of the algorithm itself?

- Pick one of the natural language algorithms, 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 mathematical expression-based pseudocode algorithms, 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 concepts and practices for writing pseudocode, which can serve as an aid to the implementation of algorithms in computer code, or as a tool for documenting the algorithmic logic of existing computer code.

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. All in all, it's quite common that two different but equally competent developers, working from the same description of an algorithm (even one written in very detailed pseudocode), using the same programming language, and following the same coding conventions, will end up with different implementations. Even with a shared starting point, each programmer's unique 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 exploration.

References

[1] Brad Arkin, Frank Hill, Scott Marks, Matt Schmid, Thomas John Walls, and Gary McGraw, “How We Learned to Cheat at Online Poker: A Study in Software Security,” Datamation, June 7, 2001. [Online]. Available: https://www.datamation.com/applications/how-to-cheat-at-online-poker-a-study-in-software-security/. [Accessed: January 27, 2025].

[2] Rob Weir, “Doing the Microsoft Shuffle: Algorithm Fail in Browser Ballot,” February 27, 2010. [Online]. Available: http://www.robweir.com/blog/2010/02/microsoft-random-browser-ballot.html. [Accessed: January 27, 2025].

[3] “Shuffling”, *Wikipedia—The Free Encyclopedia*, January 21, 2025. [Online]. Available: http://en.wikipedia.org/wiki/Shuffling#Algorithms. [Accessed: January 27, 2025].

[4] Ronald A. Fisher, Frank Yates, *Statistical tables for biological, agricultural and medical research*, 3rd ed. Longman, 1948, pp. 26–27.

[5] Richard Durstenfeld, “Algorithm 235: Random permutation”, *Communications of the ACM*, July 1964.

[6] J. J. O'Connor and E. F. Robertson, “Eratosthenes of Cyrene”, January 1999. [Online]. Available: https://mathshistory.st-andrews.ac.uk/Biographies/Eratosthenes/. [Accessed: January 27, 2025].

[7] Chris K. Caldwell, “The Prime Glossary: Sieve of Eratosthenes”, 2025. [Online]. Available: http://primes.utm.edu/glossary/xpage/SieveOfEratosthenes.html. [Accessed: January 27, 2025].

[8] Charlene Douglass, “Euclid”, *Math Open Reference*, 2011. [Online]. Available: http://www.mathopenref.com/euclid.html. [Accessed: January 27, 2025].

[9] Propositions 1 and 2, Euclid (David E. Joyce, ed.), *Elements,Book VII*, c. 300 BCE (ed. 2010). [Online]. Available: http://aleph0.clarku.edu/~djoyce/java/elements/bookVII/bookVII.html. [Accessed: January 27, 2025].

[10] “Modulo”, *Wikipedia—The Free Encyclopedia*, January 25, 2025. [Online]. Available: http://en.wikipedia.org/wiki/Modulo. [Accessed: January 27, 2025].

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

[12] George T. Heineman, Gary Pollice, and Stanley Selkow, *Algorithms in a Nutshell*, O'Reilly, 2008, pp. 169-171.

[13] Joseph L. Ganley, “Jarník-Prim algorithm”, *Programming-Algorithms.net*, August 14, 2004. [Online]. Available: http://www.programming-algorithms.net/article/43764/Jarnik-Prim-algorithm. [Accessed: January 27, 2025].

[14] “Convex Set”, *PlanetMath.org*, February 9, 2018. [Online]. Available: https://planetmath.org/convexset. [Accessed: January 27, 2025].

[15] Robert Sedgewick, *Algorithms*, 2nd ed. Addison-Wesley, 1989, pp. 359-365.

[16] Pankaj Sharma, "Gift Wrap Algorithm (Jarvis March Algorithm) to find Convex Hull", *OpenGenus IQ*, July 2, 2018. [Online]. Available: https://iq.opengenus.org/gift-wrap-jarvis-march-algorithm-convex-hull/. [Accessed: January 27, 2025].