Brief Introduction to Algorithms

What is an algorithm?

An *algorithm* is a step-by-step method for solving a problem. Think of it as a procedure whose steps are completely specified to always give the correct result for valid input. It is the *idea* behind every computer program. Examples of algorithms can be found throughout history, starting with ancient Babylonia. School children learn to compute the greatest common divisor of two integers using the Euclidean algorithm which was described by Euclid in 300 BC. Indeed, the word "algorithm" derives from the name of the ninth-century Arabic mathematician Al-Khwarizmi.

Informally, think of algorithm as just a fancy word for *recipe* or *rule of procedure*. A recipe is nothing but a sequence of steps that instruct a cook in preparation of a particular dish. The following recipe for salmon furnishes an example of an algorithm:

1. Take a $1-$ to $2-$ pound salmon and allow it to swim in clear water for 24 hours.
2. Scale and fillet the salmon.
3. Rub fillets with butter and season with salt and pepper.
4. Place on cedar board and bake in oven set at 350° for 25 minutes.
5. Throw away salmon and eat cedar board.

While a recipe describes the *ingredients* and the *sequence of steps* required to convert the ingredients into a dish, an algorithm describes the *input data* and *sequence of steps* to generate the required output from the given input. Just as there are many recipes to make a dish, a problem or class of problems may have many different algorithms. And, for each such algorithm, there can be many different implementations of computer programs.

Formally, an algorithm is a procedure to accomplish a well-specified problem and has the following characteristics:

- ***Input***. The algorithm receives zero or more inputs taken from a specified set of objects.
- ***Output***. The algorithm produces one or more output which have a relation to the inputs.
- ***Precision***. The steps are precisely and rigorously stated without cause for any ambiguity in their actions.
- ***Finiteness***. The algorithm must terminate after a finite number of steps are executed.
- ***Uniqueness***: The intermediate results of each step of execution are uniquely defined and depend only on the input to and results of the preceding steps.
- ***Generality***. To be interesting, the algorithm must solve the problem for a general set of inputs.

Finally, a word about why the subject of algorithms is important and useful. One specific reason is that algorithms are the basis of all computer programs and thus this subject underlines the entire field of computing and computer science. A more general reason is that the discipline of algorithmic thinking is valuable in carrying out any non-trivial project because it aids in the planning of such tasks and, in particular, helps avoidance of errors by forcing a consideration of all possibilities. This latter point is crucial when designing a large system such as a manned lunar probe, where forgetting even the smallest detail will lead to catastrophic results.

Common elements of algorithms

Algorithms contain the following five elements:

- *Input*: Algorithms must have the means of acquiring data from external means. For example, an algorithm that computes the roots of a quadratic equation $ax^2 + bx + c$ must receive the coefficients a, b, c as input.
- *Sequence*: Algorithms must be able to perform actions in order, one after the other. A majority of these sequential actions consist of arithmetic computations [addition, subtraction, multiplication, division, and so on], relational comparisons [greater than, less than, and so on], and testing logical conditions [equal to, not equal to, and so on].
- *Selection*: Algorithms must have some means of selecting (or choosing) among two possible courses of actions based on initial data, input values, and/or computed values.
- *Iteration*: Algorithms must be able to repeatedly execute a collection of instructions, for a fixed number of times or until some logical condition holds.
- *Output*: At their conclusion, algorithms must have the means to report the results obtained by executing the algorithm.

Expressing algorithms

The term algorithm is derived from the name of a ninth-century Arabic textbook author, Musa al-Khowarizimi (son of Moses, native of Khowarizm). It is a word familiar to - although perhaps quickly forgotten by - generations of secondary school students many of whom have learned Euclid's algorithm, a rule for computing the greatest common divisor of two integers.

Indeed, algorithms are something most people use every day even when they've never heard of the word. When using a recipe in a cookbook, the algorithm is the specific sequence of instructions which the cook follows. The route information published by mapping apps provides another example of an algorithm. In this case, the route is the specific sequence of directions that must be followed to travel from a source location to a destination location. Often algorithms are used implicitly, that is, without explicit thought about a sequence of steps. You just don't know it or think about it much. Almost every activity in your lives can be modeled as algorithm: figuring out what to wear, alphabetizing books on a shelf, folding laundry retrieved from a dryer, how to navigate the grocery store to purchase weekly supplies, determining how to prioritize your tasks for the day, and so on.

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. And to determine what these instructions are, an algorithm, apart from any specific computer and any specific programming language, must be devised to solve the problem. Algorithms intended to be executed or run or performed on a computer must avoid imprecision and ambiguities because computers do exactly what they're told and nothing more. They are notoriously poor at resolving ambiguities since they lack the ability to read between the lines and they lack the skill to determine what was meant rather than what was said. An informal instruction "please get me a glass of water" in a non-computing context can be understood and accomplished by you and most humans but not by a computer. In contrast, an algorithm is a sequence of steps that describes how to solve a problem and/or complete a task, which will always give the correct result. For the previous non-computing "please get me a glass of water" instruction, the algorithm might be

1. Go to the kitchen.
2. Pick up a glass.
3. Turn on the tap.
4. Put the glass under the running water and remove it once it is almost full.
5. Turn off the tap.
6. Take the glass back to the person who gave the instruction.

It is reasonable to assume that you and most other humans would be able to follow these steps and fetch the glass of water. However, from a computational context, no computer could execute these steps. These steps are too informal, too lacking in precision. For example, how would a computer execute the step "Go to the kitchen." Assuming the person requesting the glass of water is on the patio, how does the computer navigate its way to the kitchen? How does the computer open the door separating the patio from the kitchen. How does it avoid obstacles? What has been said about the lack of precision in the first step is true for the remaining steps.

Another example in a computational context might be to "display the list of high scores" for a video game. Again, informal instructions like this aren't precise; there's no way that a computer could follow that instruction exactly, but you would get the general idea of what is meant and what you have to do, which would be "go through each score in a table of scores for various players, keeping track of the largest score and the player's name." These sorts of informal descriptions are only useful for giving another human the general idea of what you mean, and even then there's a risk that they won't properly understand it. In a computational context, the video game's authors will have to implement the algorithm in more detail with more precise steps before a programmer could author a program that executes the algorithm.

To emphasize the idea of precisely describing actions in a computational algorithm, it is de rigueur for introductory programming courses to have students write down an algorithm to make a peanut butter and jelly (PBJ) sandwich. The instructor then executes these algorithms, which are often imprecise and ambiguous. The instructor takes the most comical interpretation of the instructions to underscore that what the students wrote did not actually describe what they meant.

This exercise underscores an important point - you must specify *exactly* what you want the computer to do. The computer does not "know what you mean" when you write something vague, nor can it figure out an "etc." Instead, you must be able to describe exactly what you want to do in a step-by-step fashion. Precisely describing the exact steps to perform a specific task is somewhat tricky, as you are used to people implicitly understanding details you omit. Irrespective of the programming language used, no computer will do that for you.

Describing and writing algorithms requires a notation for expressing a sequence of steps to be performed. The options available to us include natural languages such as English and programming languages such as Java, Python, C, C++, and so on.

Humans communicate in their daily lives using natural languages such as English, Spanish, Japanese, and so on. However, human communications are too rich, ambiguous, and depend on context. They lack structure and therefore would be hard to follow when used for expressing computer science algorithms.

A programming language is at the other end of the spectrum - it is too rigid for expressing solutions. When solving problems, you want to think at an abstract level. However, programming languages shift the emphasis from how to solve the problem to tedious details of grammar and syntax.

Rather than using natural languages or programming languages, an algorithm is better expressed using one of two tools: *pseudocode* or *flowchart*.

- Pseudocode is an English-like representation of the logical sequence of steps required to solve a problem.
- Flowchart is a pictorial representation of the same thing.

Writing pseudocode

Pseudocode involves thinking at an abstract level about the problem and using an English-like representation of the solution. *Pseudo* is a prefix that means *false*, and to *code* a problem means to put it in a programming language; therefore *pseudocode* means *false code*, or sentences that appear to have been written in a computer programming language but don't necessarily follow syntax rules of any specific language.

Pseudocode is flexible and provides a good compromise between natural and programming languages because it is a planning tool, and not the final product. It is simple, readable, and has no rigid rules to adhere to.

Any pseudocode must be able to represent the five common elements of any algorithm: *input*, *sequence*, *selection*, *iteration*, and *output*. Sometimes, pseudocode is prefaced with a beginning statement like start and ended with a terminating statement like stop. The statements between start and stop look like English and are indented slightly so that start and end stand out.

Input

Reading input from an user would be written as:

$$\begin{aligned}&\textbf{1:}\ \text{read}\ x\\&\textbf{2:}\ \text{read}\ a\ , b,\ c\end{aligned}$$

Output

Printing or presenting the output to the user can be written as:

$$\begin{aligned}&\textbf{1:}\ \text{print}\ x\\&\textbf{2:}\ \text{print}\ \text{”your speed is: ”}\ x\end{aligned}$$

Sequence

A sequence structure consists of computational statements that are executed in order, one after the other. Basic *arithmetic operations* are written as:

$$\begin{aligned}&\textbf{1:}\ x = 10\\&\textbf{2:}\ y = (ax^2 + bx + c)\\&\textbf{3:}\ z = y\end{aligned}$$

A bit of terminology first. Each computational line in the pseudocode and more generally every line of the pseudocode is labeled a *statement*. Each statement consists one or more *expressions*. Line 1 containing the text $x = 10$ is a statement. This statement consists of 3 expressions (from left to right): x, 10, and $x = 10$. Every expression consists of one or more *operands* and zero or more *operators*. Operators represent actions while operands represent the objects on which these actions are applied. In expression $x = 10$, symbol $=$ represents the operator while symbol x and value 10 represent operands. While 10 will always intrinsically describe an integer value 10, symbol x specifies a variable. Think of a variable as a *named memory location* that can store values. Unlike constant 10, the contents of a variable change during the course of execution of the program.

In terms of execution, every expression *evaluates* to a *value* and a *type* (more on types in subsequent lectures).

At line 1, statement $x = 10$ means "Copy value 10 into x." More generally, the statement

$$variable = expression$$

means "Copy value of $expression$ into $variable$," or, equivalently, "Replace the current value of $variable$ by the value of $expression$." For example, when the statement on line 3

$$z = y$$

is executed, the value of y is copied into z and the value of y is unchanged. The symbol $=$ is interpreted as "is assigned." We call symbol $=$ the *assignment operator*.

Relational expressions are used to compare two values.

$$\begin{array}{l}\textbf{1: } a = (x == y)\\ \textbf{2: } b = (x \mathrel{!}= y)\\ \textbf{3: } c = (x > y)\\ \textbf{4: } d = (x >= y)\\ \textbf{5: } e = (x < y)\\ \textbf{6: } f = (x <= y)\end{array}$$

The relational operators $==$, $!=$, $>$, $>=$, $<$, and $<=$ have the usual meanings from algebra: equal to, not equal to, greater than, greater than or equal to, less than, and less than equal to, respectively. The thing to remember though is that relational expressions evaluate to a $true$ or $false$ value. By convention, zero value 0 is considered $false$ while any non-zero value evaluates $true$.

Logical expressions are mostly used to control the sequencing of steps. They allow the pseudocode to make a decision based on multiple conditions. Each operand is considered to be a condition that can be evaluated $true$ or $false$. Then the values of the conditions are combined to determine the overall value of the groupings $operand1\ operator\ operand2$ or $!operand$ grouping.

$$\begin{array}{l}\textbf{1: } a = (x \mathrel{\&\&} y)\\ \textbf{2: } b = (x \mid\mid y)\\ \textbf{3: } c = !x\end{array}$$

Operator $\&\&$ perform a logical AND operation, that is, the expression $operand1\ \&\&\ operand2$ is $true$ only if both operands are $true$ [non-zero]. Operator $||$ performs a logical OR operation, that is, the expression $operand1\ ||\ operand2$ is $true$ when either operand is $true$ [non-zero]. Operator $!$ performs a logical NOT [negation] operation, that is $!operand$ is $true/false$ only if $operand$ is $false/true$.

Selection

Input, output, and sequence statements are sequentially executed, that is, the statements are executed in order one after the other. In contrast, a *selection statement* alters the sequential flow of control by allowing one of two courses of action to take place depending on whether the answer to a condition is $true$ or $false$. An action that might or might not be executed can be written as:

$$\begin{array}{l}\textbf{1: if } (condition\ is\ true) \textbf{ then}\\ \textbf{2: } \quad action\ to\ be\ performed\\ \textbf{3: else}\\ \textbf{4: } \quad some\ other\ action\ to\ be\ performed\\ \textbf{5: endif}\end{array}$$

Notice that the steps performed by the **if - then** and **else** clauses are indented slightly so that they can stand out and can be easily discerned. Such conditional situations arise often in our daily lives. For example:

$$
\begin{array}{l}
\textbf{1: if } (today\ is\ payday) \textbf{ then} \\
\textbf{2: } \quad dine\ in\ restaurant \\
\textbf{3: else} \\
\textbf{4: } \quad eat\ Ramen\ noodles\ at\ home \\
\textbf{5: endif}
\end{array}
$$

In certain situations, the **else** clause doesn't appear. For example, when there is a high probability that it might rain, you would like to carry an umbrella to stay dry during your commute. If rain is not forecast, you go about your commute as usual:

$$
\begin{array}{l}
\textbf{1: if } (today\ is\ forecast\ to\ be\ wet) \textbf{ then} \\
\textbf{2: } \quad carry\ umbrella \\
\textbf{3: endif } [\textbf{else} \text{ clause is not required here}] \\
\textbf{4: } [\text{continue with the usual steps involved in your commute}]
\end{array}
$$

Line 4 in the pseudocode contains text delimited by symbols [and] representing a *comment*. Comments are not an intrinsic part of an algorithm but are instead present when the algorithm's author wishes to convey explanatory information to readers of the algorithm.

Iteration operations

The fifth element represents *repeat operations* where a block of basic operations are repeatedly executed until a certain condition is met:

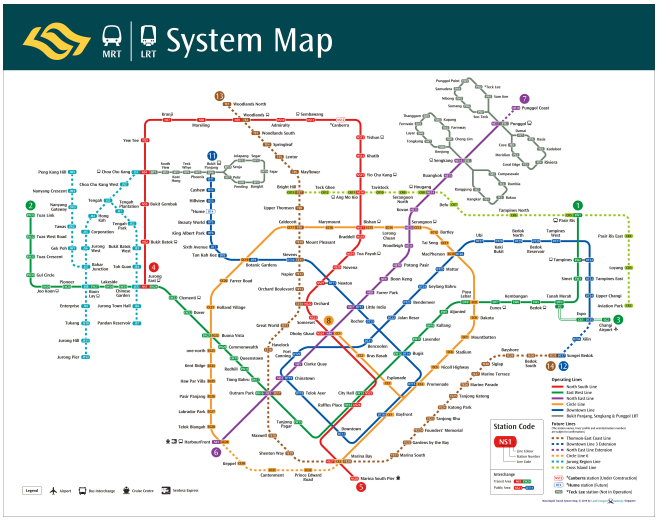
$$
\begin{array}{l}
\textbf{1: while } (condition\ is\ true) \\
\textbf{2: } \quad repeated\ statements \\
\textbf{3: endwhile} \\
\textbf{4: } [\text{steps after } \textbf{while} \text{ operations}]
\end{array}
$$

Notice that the steps repeatedly performed by the **while** statement are indented slightly so that they can stand out and can be easily discerned. How does this work? First, $condition$ is evaluated. If it is $false$, then the loop terminates and the step on line 3 immediately following the loop is executed. Instead, if $condition$ is $true$, then the algorithm executes the $repeated\ statements$ in order, one by one. Here's an example that prints a table of numbers and their corresponding squares:

$$
\begin{array}{l}
\textbf{1: } count = 1 \\
\textbf{2: while } (count\ <=\ 10) \\
\textbf{3: } \quad square = count * count \\
\textbf{4: } \quad \text{print value of } square \text{ and value of } count \\
\textbf{5: } \quad count = count\ +\ 1 \\
\textbf{6: endwhile}
\end{array}
$$

Pseudocode example: Traveling on a subway

Suppose you wish to travel from station A to station B on the Singapore Mass Rapid Transit [MRT] system. The following picture shows a map of the MRT [image from lta.gov.sg] with different lines indicating different colors.



Let's embody the thought process you might apply in deciding how to make your trip from station A to station B into an algorithm expressed in pseudocode:

Algorithm MRT

Input: Source station A and destination station B
Output: Directions to go from station A to station B

1. **if** A and B are on the same line L **then**
2. travel on line L from A to B
3. **else** [A and B are on different lines]
4. find all pairs of lines (L_i, L_j) such that A is a station on L_i and B is a station on L_j
5. [assume the two lines have a station in common]
6. **if** there is only one such pair **then**
7. take L_i to a station also on L_j and then take L_j to station B
8. **else** [more than one possible pair of lines]
9. use pair (L_i, L_j) which requires fewest stops
10. **endif**
11. **endif**

Algorithm MRT should be understandable even if you've never seen an algorithm before. You may have noticed ways that MRT might be improved. For example, suppose you know a long walk is required to change from one line to another at station C compared to some other station D. How might that fact be used by the algorithm? You should always be on the lookout for ways in which an individual algorithm may be improved.

You may have also noted some ambiguities in MRT. What do you do if two pairs of lines require the same number of stops? How about the case where two lines have more than one station in common? What if certain lines are less busy than other lines at certain parts of the day and thus it is highly likely that you'd be able to find a seat? And, what if the more comfortable journey takes more time than the less comfortable journey? How should algorithm MRT decide between the two options? These are not serious problems with MRT because it is an algorithm intended for consumption by humans who are usually pretty good at "reading between the lines" and using situational contexts to resolve ambiguities.

Pseudocode example: Mathematical algorithm

Lets consider a problem from mathematics:

$$\text{Compute sum of the first } n \text{ natural numbers of the series } \sum_{i=1}^{n} \frac{1}{i}$$

The input is the terminating index n, and the output is the sum of the first n terms of the series. If you aren't familiar with summation notation $\sum_{i=1}^{n} \frac{1}{i}$, it means that you take the quantity after the $\sum$ and evaluate it for each value of i implied by the limit $i = 1$ below the $\sum$ and the limit $i = n$ above the $\sum$ and then add the evaluated values together. Thus, for example,

$$\begin{aligned}\sum_{i=1}^{5} \frac{1}{i} &= \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} \\ &= 1.0 \ + \ 0.5 \ + \ 0.333333 \ + \ 0.25 \ + \ 0.2 \\ &= 2.283333\end{aligned}$$

The following algorithm SUM is developed to solve the problem.

Algorithm SUM

Input: positive natural number n

Output: $\frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n}$

1. $sum = 0$
2. $i = 1$
3. **while** $(i <= n)$
4. $\quad sum = sum + \frac{1}{i}$
5. $\quad i = i + 1$
6. **endwhile**
7. print value of sum

The algorithm receives a positive natural number n as input and produces as output, sum, the sum of the first n terms of the series.

The first task of the algorithm is to initialize variables to their appropriate values. Lines 1 and 2 initialize variable sum (keeping track of the summation computed so far) to 0 and counter i (keeping track of the numbers whose reciprocals are to be computed) to 1.

In line 3, the **while**'s conditional expression $i <= n$ is $true$ if i is smaller-than-or-equivalent-to n. The condition will be $false$ when i becomes greater-than n. This is the correct and necessary condition to compute $\sum_{i=1}^{n} \frac{1}{i}$ for all values of i in the interval $[1, n]$. Lines 4 and 5 are repeatedly executed to compute the summation.

To test algorithm SUM, we should check it by hand for several values of n. Simulating the execution of an algorithm for a specific input is called a *trace*. Here the minimum value of n is 1; n has no maximum since n can be equal to any positive integer.

If n is 1, at lines 1 and 2, sum and i are set to 0. At line 3, i is incremented to 1. At line 4, sum is set to 1. The algorithm terminates at line 5 since i is equal to n; both are now equal to 1. The algorithm correctly computes 1.

Next, we show how the algorithm executes if the input, n, is equal to 3. At lines 1 and 2, the algorithm sets sum to 0 and i to 0. At line 3, it assigns the value $i + 1$, which is 1 since i is 0, to i. The effect is to add 1 to i. At line 4, it adds $\frac{1}{i}$ [i.e., $\frac{1}{1}$] to sum, so that sum now has value 1. At line 5, the algorithm tests whether i equals n. Because i equals 1 and n equals 3, the algorithm repeats lines 3 and 4.

At line 3, the algorithm updates i to 2. It then adds $\frac{1}{i}$ [i.e., $\frac{1}{2}$] to sum. At this point, the value of sum is $1 + \frac{1}{2}$. Since i now equals 2 and therefore is not equal to n, the algorithm repeats lines 3 and 4.

At line 3, the algorithm updates i to 3. It then adds $\frac{1}{i}$ [i.e., $\frac{1}{3}$] to sum. At this point, the value of sum is $1 + \frac{1}{2} + \frac{1}{3}$. Since i equals 3, i is now equal to n. Therefore, the algorithm terminates. The value, stored in sum, is the sum of the first three terms of the series: 1.833333.

Recall that an algorithm must have the following characteristics: input, output, precision, finiteness, uniqueness, and generality. Let's evaluate algorithm SUM for these characteristics:

- *Input*. The algorithm receives exactly one input - the terminating index n - which can have any of the infinite values of a positive natural number.
- *Output*. The algorithm produces the required summation $\sum_{i=1}^{n} \frac{1}{i}$.
- *Precision*. The algorithm is decomposed into simple sequential steps from lines 1 through 7 each of which performs a specific action without any ambiguity.
- *Finiteness*. This is where most algorithms have errors - it is especially important to test *boundary values*: the minimum and maximum values. First, the counter i is initialized to 1 on line 3. If $n == 1$ [that is, n is equivalent to 1], the algorithm stops after one repetition since line 5 increments i to 2 ensuring that the condition on line 3 will evaluate $false$. There is no maximum value for this algorithm. However large the value of n is, i is always incremented by 1 each repetition meaning that after n repetitions i will have a value of $n + 1$ at which point the condition on line 3 will evaluate $false$.
- *Generality*. The algorithm works for all positive natural numbers of which an infinite number exist.

Checking algorithm SUM for small values of n doesn't prove that the algorithm is correct. It proves only that the algorithm is correct for the values that were tested! However, checking the algorithm for small values of n does give you confidence that the algorithm is correct. To prove that the algorithm is correct, a mathematical technique such as mathematical induction could be used. Such correctness proofs are beyond the scope of this basic introduction.

Drawing flowcharts

A *flowchart* is a combination of graphical symbols that are used to represent the logical flow of data through a solution to a problem. When you create a flowchart, you draw geometric shapes that contain the individual statements and that are connected with arrows.

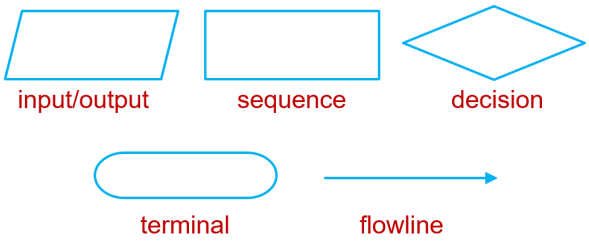
A *parallelogram* represents an input symbol indicating that it contains input operations. The input statement in English is written inside the parallelogram. The same symbol is used to represent an output operation. Since the parallelogram is used to represent both input and output, it is often called the input/output symbol, or I/O symbol.

A *rectangle* represents sequence operations.

To implement selection and iteration structures, a decision symbol, which is shaped like a *diamond* is used. The diamond usually contains a question, the answer to which is one of two mutually exclusive options: $true$ or $false$.

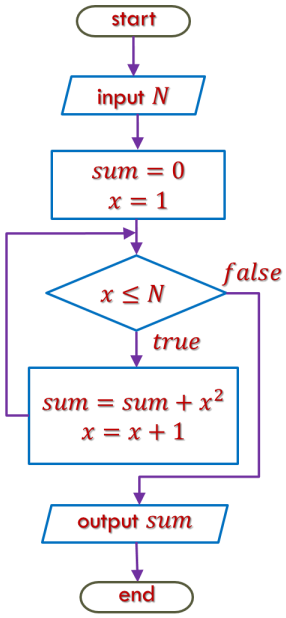
To show the correct sequence of these statements, *flowlines* represented by arrows are used to connect these graphical symbols.

To be complete, a flowchart should include the two terminal symbols at each end: *start* and *stop* symbols. These symbols are graphically represented by the lozenge shape.



Flowchart example: Computing series summation

As a first example of flowcharting an algorithm, algorithm SUM [which computes the summation $\sum_{i=1}^{n} \frac{1}{i}$] is illustrated using a flowchart.



Flowcharts are difficult to draw for complex algorithms compared to writing pseudocode. diagrams.net is a free online tool for rendering flowcharts.

Algorithm example: Counting

This basic introduction to writing algorithms concludes by writing pseudocode and drawing a flowchart for the really simple problem of counting. That is, given a non-negative integer, what is the next integer in sequence? Now this is so familiar and trivial to you that your first reaction may be: why is an algorithm required at all for such a trivial process? The rationale is that the problem here is not how to count - of course, you know how to do that - but rather whether you can explain how to count to something inanimate like a computer which doesn't know how to count or to someone who doesn't know how to count. Think about how you would explain counting to something rudimentary that its only arithmetic capability is to be able to add 1 to the digits $0, 1, \ldots, 8$ [but not 9 since adding 1 to 9 involves a carry into the 10's place]. Being able to take

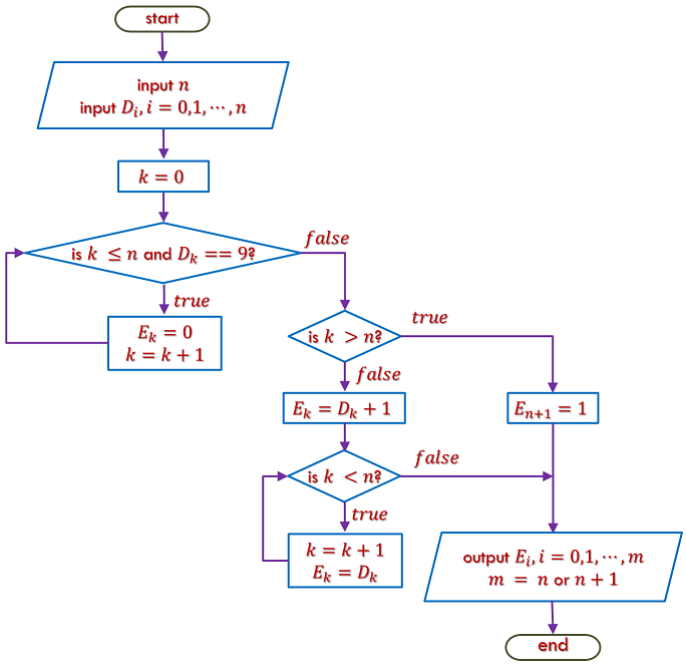
something as trivial as counting and writing an algorithm to perform that task would be a true test of your understanding of the concept of counting. Assume that an integer can be represented as a sequence of digits $I = D_n D_{n-1} D_{n-2} \dots D_1 D_0$, so that, for example, the number 6789 is represented by $D_3 = 6, D_2 = 7, D_1 = 8, D_0 = 9$ with n having a value of 3. Likewise, the number 5 is represented by $D_0 = 5$ with n having a value of 0.

Algorithm COUNTING

Input: n	$[\text{integer } \geq 0]$
$D_i, i = 0, 1, \dots, n$	$[\text{digits of } I = D_n D_{n-1} \dots D_1 D_0]$
Output: $E_i, i = 0, 1, \dots, m$	$[I + 1;\ m = n \text{ or } n + 1]$
1. $k = 0$	[initialize digit index to 0]
2. **while** $(k \leq n$ **and** $D_k == 9)$	[find first digit not 9]
3. $E_k = 0$	[change 9's to 0's]
4. $k = k + 1$	
6. **endwhile**	
7. **if** $(k > n)$ **then**	
8. $E_{n+1} = 1$	[I all 9's]
9. **else**	
10. $E_k = D_k + 1$	[increase first non-9 digit by 1]
11. **while** $(k < n)$	[copy remaining digits]
12. $k = k + 1$	
13. $E_k = D_k$	
14. **endwhile**	
15. **endif**	

Test the algorithm for general values such as the number 426, the number 429, and 942, for all of which $n = 2$. Test boundary conditions such as 2 where $n = 0$, or 9 where $n = 0$, or 99 with $n = 1$ and 999 with $n = 2$.

The following figure illustrates the flowchart corresponding to algorithm COUNTING that adds 1 to an integer $I = D_n D_{n-1} D_{n-2} \dots D_1 D_0$.



Do you prefer the pseudocode or the flowchart? If this is your first introduction to both these techniques, then I'd hazard a guess that you prefer the flowchart because it contains easy-to-follow symbols. This notation was popular in the early days of computing but is no longer used because it is quite unwieldy for all but quite small algorithms. Once you get used to pseudocode notation, you'll find it easier to read and understand than flowcharts.

Exercises

The following problems are meant for exercising your knowledge of writing pseudocode and flowcharting algorithms. The practice gained from developing algorithms, pseudocode, and flowcharting will help you write more robust and elegant code. For each problem, write the pseudocode and draw a flowchart for the algorithm that solves the problem. Before attempting to write any algorithm, you should think carefully about precisely what it is you want to do and how you're going to go about doing it.

1. This is a classic problem introduced to every computer science student. Write an algorithm to make a peanut butter and jelly sandwich.
2. In the same vein as the previous question, think about the process of calling your friend using a telephone. Next, provide an algorithm that can be followed by an inanimate robot to make a call using a telephone.
3. Compute the area of a trapezoid.
4. Compute the factorial $n!$ of a positive integer n:
   $n! = 1 \times 2 \times 3 \times \ldots \times (n-2) \times (n-1) \times n$.
5. Write an algorithm that computes the sum of the first n terms of the series $\sum_{i=1}^{\infty} \frac{1}{(1+i)^2}$.
6. Compute the greatest common divisor of two positive integers.
7. Find the roots of a quadratic equation $ax^2 + bx + c = 0$ for some real numbers a, b, and c.
8. Let $S = \{s_0, s_1, \ldots, s_{n-1}\}$ be a set of n elements where each element is a real number. Find the maximum number in the set S. The input is the sequence, S, and the integer, n, indicating the number of elements in the sequence. The output is the index of the smallest element in the sequence.
9. Alternatively, determine the minimum value in the set S from the previous question.
10. Find the length of the longest sequence of values that are in strictly increasing order given a sequence of n real values $S = \{s_0, s_1, \ldots, s_{n-1}\}$. From the definitions of increasing and strictly increasing functions, we can conclude that the sequence of values $\{2, 3, 4, 5\}$ are in strictly increasing order while the sequence of values $\{2, 2, 3, 3, 5\}$ are in increasing order. The length of the longest sequence of values in strictly increasing order is 4 for the sequence of values $\{2, 3, 4, 5\}$. The length of the longest sequence of values in $\{2, 2, 3, 3, 5\}$ is 2 - the subsets $\{2, 3\}$ and $\{3, 5\}$ are in strictly increasing order.