In programming, algorithm is a set of well defined instructions in sequence to solve the problem.

**Qualities of a good algorithm**

1. Input and output should be defined precisely.
2. Each steps in algorithm should be clear and unambiguous.
3. Algorithm should be most effective among many different ways to solve a problem.
4. An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.

**Examples Of Algorithms In Programming**

**Write an algorithm to add two numbers entered by user.**

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

sum←num1+num2

Step 5: Display sum

Step 6: Stop

**Write an algorithm to find the largest among three different numbers entered by user.**

Step 1: Start

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If a>b

If a>c

Display a is the largest number.

Else

Display c is the largest number.

Else

If b>c

Display b is the largest number.

Else

Display c is the greatest number.

Step 5: Stop

**Write an algorithm to find all roots of a quadratic equation ax2+bx+c=0.**

Step 1: Start

Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;

Step 3: Calculate discriminant

D←b2-4ac

Step 4: If D≥0

r1←(-b+√D)/2a

r2←(-b-√D)/2a

Display r1 and r2 as roots.

Else

Calculate real part and imaginary part

rp←b/2a

ip←√(-D)/2a

Display rp+j(ip) and rp-j(ip) as roots

Step 5: Stop

**Write an algorithm to find the factorial of a number entered by user.**

Step 1: Start

Step 2: Declare variables n,factorial and i.

Step 3: Initialize variables

factorial←1

i←1

Step 4: Read value of n

Step 5: Repeat the steps until i=n

5.1: factorial←factorial\*i

5.2: i←i+1

Step 6: Display factorial

Step 7: Stop

**Write an algorithm to check whether a number entered by user is prime or not.**

Step 1: Start

Step 2: Declare variables n,i,flag.

Step 3: Initialize variables

flag←1

i←2

Step 4: Read n from user.

Step 5: Repeat the steps until i<(n/2)

5.1 If remainder of n÷i equals 0

flag←0

Go to step 6

5.2 i←i+1

Step 6: If flag=0

Display n is not prime

else

Display n is prime

Step 7: Stop

**Write an algorithm to find the Fibonacci series till term≤1000.**

Step 1: Start

Step 2: Declare variables first\_term,second\_term and temp.

Step 3: Initialize variables first\_term←0 second\_term←1

Step 4: Display first\_term and second\_term

Step 5: Repeat the steps until second\_term≤1000

5.1: temp←second\_term

5.2: second\_term←second\_term+first term

5.3: first\_term←temp

5.4: Display second\_term

Step 6: Stop

Algorithm is not the computer code. Algorithm are just the instructions which gives clear idea to you idea to write the computer code.

# **Ostrich algorithm**

In [computer science](https://en.wikipedia.org/wiki/Computer_science), the **ostrich algorithm** is a strategy of ignoring potential problems on the basis that they may be exceedingly rare. It is named for the [ostrich effect](https://en.wikipedia.org/wiki/Ostrich_effect) which is defined as "to stick one's head in the sand and pretend there is no problem". It is used when it is more cost-effective to allow the problem to occur than to attempt its prevention.

## Use with deadlocks[[edit](https://en.wikipedia.org/w/index.php?title=Ostrich_algorithm&action=edit&section=1)]

This approach may be used in dealing with [deadlocks](https://en.wikipedia.org/wiki/Deadlock) in [concurrent programming](https://en.wikipedia.org/wiki/Concurrent_programming) if they are believed to be very rare and the cost of detection or prevention is high. For example, if each PC deadlocks once per 10 years, the one reboot may be less painful than the restrictions needed to prevent it.[[1]](https://en.wikipedia.org/wiki/Ostrich_algorithm#cite_note-1)

A set of processes is [deadlocked](https://en.wikipedia.org/wiki/Deadlock) if each process in the set is waiting for an event that only another process in the set can cause. Usually the event is release of a currently held resource and none of the processes can run, release resources, and be awakened.[[2]](https://en.wikipedia.org/wiki/Ostrich_algorithm#cite_note-2)

The ostrich algorithm pretends there is no problem and is reasonable to use if deadlocks occur very rarely and the cost of their prevention would be high. The [UNIX](https://en.wikipedia.org/wiki/UNIX) and [Windows](https://en.wikipedia.org/wiki/Windows) operating systems take this approach.[[3]](https://en.wikipedia.org/wiki/Ostrich_algorithm#cite_note-3)

Although using the ostrich algorithm is one of the methods of dealing with [deadlocks](https://en.wikipedia.org/wiki/Deadlock), other effective methods exist such as dynamic avoidance, [banker's algorithm](https://en.wikipedia.org/wiki/Banker%27s_algorithm), detection and recovery, and prevention.[[4]](https://en.wikipedia.org/wiki/Ostrich_algorithm#cite_note-4)

<https://www.youtube.com/watch?v=g5rJwMQtDaY>

# DEADLOCKS

Computer systems are full of resources that can only be used by one process at a time. Common examples include printers, tape drives, and slots in the system’s internal tables. Having two processes simultaneously writing to the printer leads to gibberish. Having two processes using the same file system table slot will invariably lead to a corrupted file system. Consequently, all operating systems have the ability to (temporarily) grant a process exclusive access to certain resources.

For many applications, a process needs exclusive access to not one resource, but several. Suppose, for example, two processes each want to record a scanned document on a CD. Process *A* requests permission to use the scanner and is granted it. Process *B* is programmed differently and requests the CD recorder first and is also granted it. Now *A* asks for the CD recorder, but the request is denied until *B* releases it. Unfortunately, instead of releasing the CD recorder *B* asks for the scanner. At this point both processes are blocked and will remain so forever. This situation is called a **deadlock**.

Deadlocks can also occur across machines. For example, many offices have a local area network with many computers connected to it. Often devices such as scanners, CD recorders, printers, and tape drives are connected to the network as shared resources, available to any user on any machine. If these devices can be reserved remotely (i.e., from the user’s home machine), the same kind of deadlocks can occur as described above. More complicated situations can cause deadlocks involving three, four, or more devices and users.

Deadlocks can occur in a variety of situations besides requesting dedicated I/O devices. In a database system, for example, a program may have to lock several records it is using, to avoid race conditions. If process *A* locks record *R1* and process *B* locks record *R2*, and then each process tries to lock the other one’s record, we also have a deadlock. Thus deadlocks can occur on hardware resources or on software resources.

In this chapter, we will look at deadlocks more closely, see how they arise, and study some ways of preventing or avoiding them. Although this material is about deadlocks in the context of operating systems, they also occur in database systems and many other contexts in computer science, so this material is actually applicable to a wide variety of multiprocess systems. A great deal has been written about deadlocks. Two bibliographies on the subject have appeared in *Operating Systems Review* and should be consulted for references (Newton, 1979; and Zobel, 1983). Although these bibliographies are old, most of the work on deadlocks was done well before 1980, so they are still useful.

## 3.1 RESOURCES

Deadlocks can occur when processes have been granted exclusive access to devices, files, and so forth. To make the discussion of deadlocks as general as possible, we will refer to the objects granted as **resources**. A resource can be a hardware device (e.g., a tape drive) or a piece of information (e.g., a locked record in a database). A computer will normally have many different resources that can be acquired. For some resources, several identical instances may be available, such as three tape drives. When several copies of a resource are available, any one of them can be used to satisfy any request for the resource. In short, a resource is anything that can be used by only a single process at any instant of time.

### 3.1.1 Preemptable and Nonpreemptable Resources

Resources come in two types: preemptable and nonpreemptable. A preemptable resource is one that can be taken away from the process owning it with no ill effects. Memory is an example of a preemptable resource. Consider, for example, a system with 32 MB of user memory, one printer, and two 32-MB processes that each want to print something. Process *A* requests and gets the printer, then starts to compute the values to print. Before it has finished with the computation, it exceeds its time quantum and is swapped out.

Process *B* now runs and tries, unsuccessfully, to acquire the printer. Potentially, we now have a deadlock situation, because *A* has the printer and *B* has the memory, and neither can proceed without the resource held by the other. Fortunately, it is possible to preempt (take away) the memory from *B* by swapping it out and swapping *A* in. Now *A* can run, do its printing, and then release the printer. No deadlock occurs.

A **nonpreemptable resource**, in contrast, is one that cannot be taken away from its current owner without causing the computation to fail. If a process has begun to burn a CD-ROM, suddenly taking the CD recorder away from it and giving it to another process will result in a garbled CD, CD recorders are not preemptable at an arbitrary moment.

In general, deadlocks involve nonpreemptable resources. Potential deadlocks that involve preemptable resources can usually be resolved by reallocating resources from one process to another. Thus our treatment will focus on nonpreemptable resources.

The sequence of events required to use a resource is given below in an abstract form.

1. Request the resource.
2. Use the resource.
3. Release the resource.

If the resource is not available when it is requested, the requesting process is forced to wait. In some operating systems, the process is automatically blocked when a resource request fails, and awakened when it becomes available. In other systems, the request fails with an error code, and it is up to the calling process to wait a little while and try again.

A process whose resource request has just been denied will normally sit in a tight loop requesting the resource, then sleeping, then trying again. Although this process is not blocked, for all intents and purposes, it is as good as blocked, because it cannot do any useful work. In our further treatment, we will assume that when a process is denied a resource request, it is put to sleep.

The exact nature of requesting a resource is highly system dependent. In some systems, a request system call is provided to allow processes to explicitly ask for resources. In others, the only resources that the operating system knows about are special files that only one process can have open at a time. These are opened by the usual open call. If the file is already in use, the caller is blocked until its current owner closes it.