3. **Fundamentals of Algorithms** (06 Periods)
Exchanging value of variables, counting numbers, Summation of set of numbers, Factorial computations, Fibonacci number, Reverse of Digits.

Algorithm 2.1
EXCHANGING THE VALUES OF TWO VARIABLES

Problem

Given two variables, a and b, exchange the values assigned to them.

Algorithm development

The problem of interchanging the values associated with two variables involves a very fundamental mechanism that occurs in many sorting and data manipulation algorithms. To define the problem more clearly we will examine a specific example.

Consider that the variables a and b are assigned values as outlined below. That is,

Starting configuration

a	b
721	463

This means that memory cell or variable a contains the value 721, and memory cell or variable b contains the value 463. Our task is to replace the contents of a with 463, and the contents of b with 721. In other words we want to end up with the configuration below:

Target configuration

a	b
463	721

To change the value of a variable we can use the assignment operator. Because we want a to assume the value currently belonging to b, and b the value belonging to a we could perhaps make the exchange with the following assignments:

$a := b;$ (1)
$b := a$ (2)

where ":=" is the assignment operator. In (1) ":=" causes the value stored in memory cell b to be *copied* into memory cell a.

Let us work through these two steps to make sure they have the desired effect.

We started out with the configuration

a	b
721	463

then after execution of the assignment $a := b$ we have

a	b
463	463

The assignment (1) has *changed* the value of a but has left the value of b untouched. Checking with our target configuration we see that a has assumed the value 463 as required. So far so good! We must also check on b. When the assignment step (2) i.e. $b := a$ is made *after* executing step (1) we end up with:

a	b
463	463

In executing step (2) *a is not changed* while *b* takes on the value that *currently* belongs to *a*. The configuration that we have ended up with does not represent the solution we are seeking. The problem arises because in making the assignment:

$$a := b$$

we have lost the value that *originally* belonged to *a* (i.e. 721 has been lost). It is this value that we want *b* to finally assume. Our problem must therefore be stated more carefully as:

new value of $a :=$ old value of b;
new value of $b :=$ old value of a

What we have done with our present proposal is to make the assignment

new value of $b :=$ new value of a

In other words when we execute step (2) we are *not* using the value *a* that will make things work correctly—because *a* has already *changed*.

To solve this exchange problem we need to find a way of not destroying "*the old value of a*" when we make the assignment

$$a := b$$

A way to do this is to introduce a temporary variable *t* and *copy* the original value of *a* into this variable before executing step (1). The steps to do this are:

$$t := a;$$
$$a := b$$

After these two steps we have

a	*t*	*b*
463	721	463

We are better off than last time because now we still have the old value of *a* stored in *t*. It is this value that we need for assignment to *b*. We can therefore make the assignment

$$b := t$$

After execution of this step we have:

a	*t*	*b*
463	721	721

Rechecking with our target configuration we see that *a* and *b* have now been interchanged as required.

The exchange procedure can now be outlined.

Algorithm description

1. Save the original value of *a* in *t*.
2. Assign to *a* the original value of *b*.
3. Assign to *b* the original value of *a* that is stored in *t*.

The exchange mechanism as a programming tool is most usefully implemented as a procedure that accepts two variables and returns their exchanged values.

Algorithm 2.2
COUNTING

Problem

Given a set of n students' examination marks (in the range 0 to 100) make a count of the number of students that passed the examination. A pass is awarded for all marks of 50 and above.

Algorithm development

Counting mechanisms are very frequently used in computer algorithms. Generally a count must be made of the number of items in a set which possess some particular property or which satisfy some particular condition or conditions. This class of problems is typified by the "examination marks" problem.

As a starting point for developing a computer algorithm for this problem we can consider how we might solve a particular example by hand.

Suppose that we are given the set of marks

55, 42, 77, 63, 29, 57, 89

In more detail we have:

	Marks	*Counting details for passes*
	55	previous count = 0 current count = 1
Order in	42	previous count = 1 current count = 1
which marks	77	previous count = 1 current count = 2
are	63	previous count = 2 current count = 3
examined	29	previous count = 3 current count = 3
	57	previous count = 3 current count = 4
	89	previous count = 4 current count = 5

$\therefore$ Number of students passed = 5

After each mark has been processed the current count reflects the number of students that have passed in the marks list so far encountered.

We must now ask, how can the counting be achieved? From our example above we see that every time we need to increase the count we build on the previous value. That is,

$$current\_count = previous\_count + 1$$

When, for example, we arrive at mark 57, we have

$$previous\_count = 3$$

Current_count therefore becomes 4. Similarly when we get to the next mark (i.e. 89) the *current_count* of 4 must assume the role of *previous_count*. This means that whenever a new *current_count* is generated it must then assume the role of *previous_count* before the next mark is considered. The two steps in this process can be represented by

$$current\_count := previous\_count + 1 \qquad (1)$$
$$previous\_count := current\_count \qquad (2)$$

These two steps can be repeatedly applied to obtain the count required. In conjunction with the conditional test and input of the next mark we execute step (1), followed by step (2), followed by step (1), followed by step (2) and so on.

Because of the way in which *previous_count* is employed in step (1) we can substitute the expression for *previous_count* in step (2) into step (1) to obtain the simpler expression

current_count := *current_count* + 1
↑ (new value) ↑ (old value)

The *current_count* on the RHS (right-hand side) of the expression assumes the role of *previous_count*. As this statement involves an assignment rather than an equality (which would be impossible) it is a valid computer statement. What it describes is the fact that the *new value* of *current_count* is obtained by adding 1 to the old value of *current_count*.

Viewing the mechanism in this way makes it clear that the existence of the variable *previous_count* in its own right is unnecessary. As a result we have a simpler counting mechanism.

The essential steps in our pass-counting algorithm can therefore be summarized as:

while less then *n* marks have been examined do

(a) read next mark,
(b) if current mark satisfies pass requirement then add one to count.

Before any marks have been examined the count must have the value zero. To complete the algorithm the input of the marks and the output of the number of passes must be included. The detailed algorithm is then as described below.

Algorithm description

1. Prompt then read the number of marks to be processed.
2. Initialize count to zero.
3. While there are still marks to be processed repeatedly do
   (a) read next mark,
   (b) if it is a pass (i.e. $\geqslant 50$) then add one to count.
4. Write out total number of passes.

Algorithm 2.3
SUMMATION OF A SET OF NUMBERS

Problem

Given a set of n numbers design an algorithm that adds these numbers and returns the resultant sum. Assume n is greater than or equal to zero.

Algorithm development

One of the most fundamental things that we are likely to do with a computer is to add a set of n numbers. When confronted with this problem in the absence of a computer we simply write the numbers down one under the other and start adding up the right-hand column. For example, consider the addition of 421, 583 and 714.

$$\begin{array}{r} 421 \\ 583 \\ \underline{714} \\ \cdots\ 8 \end{array}$$

The simplest way that we can instruct the computer's arithmetic unit to add a set of numbers is to write down an expression that specifies the addition we wish to be performed. For our three numbers mentioned previously we could write

$$s := 421+583+714 \qquad (1)$$

The assignment operator causes the value resulting from the evaluation of the right-hand side of statement (1) to be placed in the memory cell allocated to the variable s.

Expression (1) will add three *specific* numbers as required. Unfortunately it is capable of doing little else. Suppose we wanted to sum three other numbers. For this task we would need a new program statement.

It would therefore seem reasonable that all constants in expression (1) could be replaced by variables. We would then have

$$s := a+b+c \qquad (2)$$

sets of three numbers.

A fundamental goal in designing algorithms and implementing programs is to make the programs general enough so that they will successfully handle a wide variety of input conditions. That is, we want a program that will add any n numbers where n can take on a wide range of values.

The approach we need to take to formulate an algorithm to add n numbers in a computer is different from what we would do conventionally to solve the problem. Conventionally we could write the general equation

$$s = (a_1+a_2+a_3+ \cdots +a_n) \qquad (3)$$

or equivalently

$$s = \sum_{i=1}^{n} a_i \qquad (4)$$

(*Reminder*: $\sum$ is the mathematical summation operator)

One way to do this that takes note of the fact that the computer adds two numbers at a time is to start by adding the first two numbers a_1 and a_2. That is,

$$s := a_1 + a_2 \qquad (1)$$

We could then proceed by adding a_3 to the s computed in step (1).

$$s := s + a_3 \qquad (2) \quad \text{(cf. counting statement in algorithm 2.2)}$$

In a similar manner:

$$\left.\begin{array}{l} s := s + a_4 \\ s := s + a_5 \\ \vdots \quad \vdots \quad \vdots \\ s := s + a_n \end{array}\right\} \qquad (3, \ldots, n-1)$$

From step (2) onwards we are actually repeating the same process over and over—the only difference is that values of a and s change with each step. For general i^{th} step we have

$$s := s + a_{i+1} \qquad (i)$$

This general step can be placed in a loop to iteratively generate the sum of n numbers.

The core of the algorithm for summing n numbers therefore involves a special step followed by a set of n iterative steps. That is,

1. Compute first sum ($s = 0$) as special case.
2. Build each of the n remaining sums from its predecessor by an iterative process.
3. Write out the sum of n numbers.

Algorithm description

1. Prompt and read in the number of numbers to be summed.
2. Initialize sum for zero numbers.
3. While less than n numbers have been summed repeatedly do
   (a) read in next number,
   (b) compute current sum by adding the number read to the most recent sum.
4. Write out sum of n numbers.

Algorithm 2.4
FACTORIAL COMPUTATION

Problem

Given a number n, compute n factorial (written as $n!$) where $n \geq 0$.

Algorithm development

We can start the development of this algorithm by examining the definition

of $n!$. We are given that

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n \qquad \text{for } n \geq 1$$

and by definition

$$0! = 1$$

In formulating our design for this problem we need to keep in mind that the computer's arithmetic unit can only multiply *two* numbers at a time.

Applying the factorial definition we get

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \times 1 \\ 2! &= 1 \times 2 \\ 3! &= 1 \times 2 \times 3 \\ 4! &= 1 \times 2 \times 3 \times 4 \\ &\vdots \end{aligned}$$

We see that 4! contains *all* the factors of 3!. The only difference is the inclusion of the number 4. We can generalize this by observing that $n!$ can always be obtained from $(n-1)!$ by simply multiplying it by n (for $n \geq 1$). That is,

$$n! = n \times (n-1)! \qquad \text{for } n \geq 1$$

Using this definition we can write the first few factorials as:

$$\begin{aligned} 1! &= 1 \times 0! \\ 2! &= 2 \times 1! \\ 3! &= 3 \times 2! \\ 4! &= 4 \times 3! \\ &\vdots \end{aligned}$$

If we start with $p = 0! = 1$ we can rewrite the first few steps in computing $n!$ as:

$p := 1$	(1)	$= 0!$
$p := p*1$	(2 ... $n+1$)	$= 1!$
$p := p*2$		$= 2!$
$p := p*3$		$= 3!$
$p := p*4$		$= 4!$
$\vdots$		$\vdots$

From step (2) onwards we are actually repeating the same process over and over. For the general $(i+1)^{th}$ step we have

$$p := p*i \qquad (i+1)$$

This general step can be placed in a loop to iteratively generate $n!$. This allows us to take advantage of the fact that the computer's arithmetic unit can only multiply two numbers at a time.

In many ways this problem is very much like the problem of summing a set of n numbers (algorithm 2.3). In the summation problem we performed a

set of additions, whereas in this problem we need to generate a set of products. It follows from the general $(i+1)^{th}$ step that all factorials for $n \geq 1$ can be generated iteratively. The instance where $n = 0$ is a special case which must be accounted for directly by the assignment

$$p := 1 \qquad \text{(by definition of 0!)}$$

The central part of the algorithm for computing $n!$ therefore involves a special initial step followed by n iterative steps.

1. Treat 0! as a special case ($p := 1$).
2. Build each of the n remaining products p from its predecessor by an iterative process.
3. Write out the value of n factorial.

Algorithm description

1. Establish n, the factorial required where $n \geq 0$.
2. Set product p for 0! (special case). Also set product count to zero.
3. While less than n products have been calculated repeatedly do
   (a) increment product count,
   (b) compute the i^{th} product p by multiplying i by the most recent product.
4. Return the result $n!$.

Algorithm 2.6
GENERATION OF THE FIBONACCI SEQUENCE

Problem

Generate and print the first n terms of the Fibonacci sequence where $n \geq 1$. The first few terms are:

0, 1, 1, 2, 3, 5, 8, 13, ...

Each term beyond the first two is derived from the sum of its two nearest predecessors.

Algorithm development

From the definition we are given that:

new term = preceding term + term before preceding term

The last sentence of the problem statement suggests we may be able to use the definition to generate consecutive terms (apart from the first two) iteratively.

Let us define:

- a as the *term before the preceding term*
- b as the *preceding term*
- c *new term*

Then to start with we have:

	$a := 0$	first Fibonacci number
	$b := 1$	second Fibonacci number
and	$c := a+b$	third Fibonacci number (from definition)

Algorithm description

1. Prompt and read n, the number of Fibonacci numbers to be generated.
2. Assign first two Fibonacci numbers a and b.
3. Initialize count of number generated.
4. While less than n Fibonacci numbers have been generated do
   (a) write out next two Fibonacci numbers;
   (b) generate next Fibonacci number keeping a relevant;
   (c) generate next Fibonacci number from most recent pair keeping b relevant for next computation;
   (d) update count of number of Fibonacci numbers generated, i.
5. If n even then write out last two Fibonacci numbers else write out second last Fibonacci number.

Algorithm 2.7
REVERSING THE DIGITS OF AN INTEGER

Problem

Design an algorithm that accepts a positive integer and reverses the order of its digits.

Algorithm development

Digit reversal is a technique that is sometimes used in computing to remove bias from a set of numbers. It is important in some fast information-retrieval algorithms. A specific example clearly defines the relationship of the input to the desired output. For example,

Input: 27953
Output: 35972

We can get the number 2795 by integer division of the original number by 10

i.e. 27953 **div** 10→2795

This chops off the 3 but does not allow us to save it. However, 3 is the remainder that results from dividing 27953 by 10. To get this remainder we can use the **mod** function. That is,

27953 **mod** 10→3

Therefore if we apply the following two steps

$r := n$ **mod** 10 (1)⇒($r = 3$)
$n := n$ **div** 10 (2)⇒($n = 2795$)

we get the digit 3, and the new number 2795. Applying the same two steps to the new value of n we can obtain the 5 digit. We now have a mechanism for iteratively accessing the individual digits of the input number.

Our next major concern is to carry out the digit reversal. When we apply our digit extraction procedure to the first two digits we acquire first the 3 and then 5. In the final output they appear as:

3 followed by 5 (or 35)

If the original number was 53 then we could obtain its reverse by first extracting the 3, multiplying it by 10, and then adding 5 to give 35. That is,

$3 \times 10 + 5 \rightarrow 35$

The last three digits of the input number are 953. They appear in the "reversed" number as 359. Therefore at the stage when we have the 35 and then extract the 9 we can obtain the sequence 359 by multiplying 35 by 10 and adding 9. That is,

$$35\times10+9\rightarrow359$$

Similarly

$$359\times10+7\rightarrow3597$$

and

$$3597\times10+2\rightarrow35972$$

Rewriting the multiplication and addition process we have just described in terms of the variable *dreverse* we get

	Iteration	*Value of dreverse*
[1]	*dreverse* := *dreverse**10+3	3
[2]	*dreverse* := *dreverse**10+5	35
[3]	*dreverse* := *dreverse**10+9	359
	⋮	⋮

Therefore to build the reversed integer we can use the construct:

dreverse := (previous value of *dreverse*)*10
+(most recently extracted rightmost digit)

The variable *dreverse* can be used on *both* sides of this expression. For the value of *dreverse* to be correct (i.e. *dreverse* = 3) after the first iteration it must initially be zero. This initialization step for *dreverse* is also needed to ensure that the algorithm functions correctly when the input number to be reversed is zero.

Algorithm description

1. Establish *n*, the positive integer to be reversed.
2. Set the initial condition for the reversed integer *dreverse*.
3. While the integer being reversed is greater than zero do
   (a) use the remainder function to extract the rightmost digit of the number being reversed;
   (b) increase the previous reversed integer representation *dreverse* by a factor of 10 and add to it the most recently extracted digit to give the current *dreverse* value;
   (c) use integer division by 10 to remove the rightmost digit from the number being reversed.

Chapter 3
FACTORING METHODS

Algorithm 3.1
FINDING THE SQUARE ROOT OF A NUMBER

Problem

Given a number m devise an algorithm to compute its square root.

Algorithm development

When initially confronted with the problem of designing an algorithm to compute square roots, we may be at a loss as to just where to start. In these circumstances we need to be really sure of what is meant by "the square root of a number". Taking some specific examples, we know that the square root of 4 is 2, the square root of 9 is 3, and the square root of 16 is 4 and so on. That is,

$$\begin{array}{l} 2\times 2=4 \\ 3\times 3=9 \\ 4\times 4-16 \\ \vdots\ \ \vdots\ \ \vdots \end{array}$$

From these examples we can conclude that in the general case the square root n, of another number m must satisfy the equation

$$n\times n=m \qquad (1)$$

To try to make progress towards a better algorithm, let us again return to the problem of finding the square root of 36. In choosing 9 as our initial guess, we found that

$9^2=81$ which is *greater than* 36.

We know from equation (1) that the 9 should divide into 36 to give a quotient of 9 if it is truly the square root. Instead 9 divides into 36 to give 4. Had we initially chosen 4 as our square root candidate, we would have found

$4^2=16$ which is *less than* 36.

From this we can see that when we choose a square root candidate that is too large, we can readily derive from it another candidate that is too small. The larger the guess is that is too large, the correspondingly smaller will be the guess that is too small. In other words, the 9 and the 4 tend to cancel out each other by deviating from the square m in opposite directions. Thus,

Square	Square Root
81	9
36	??
16	4

The square root of 36 must lie somewhere between 9, which is too big, and 4, which is too small. Taking the average of 9 and 4:

$$\frac{(9+4)}{2} = 6.5$$

gives us an estimate "in between" 9 and 4. This new estimate may again be either greater than 36, equal to, or less than 36. We find that $6.5^2 = 42.25$ which is greater than 36. Dividing this new value into 36:

$$\frac{36}{6.5} = 5.53$$

we see that it again has a complementary value (i.e. 5.53) that is less than 36. Thus,

Square		Square Root
81	greater	9
42.25	than 36	6.5
36		??
30.5809	less	5.53
16	than 36	4

We now have two estimates of the square root, one on either side, that are closer than our first two estimates.

We can proceed to get an even better estimate of the square root by averaging these two most recent guesses:

$$(6.5+5.53)/2 = 6.015$$

Our first task now is to clarify the averaging rule that we intend to use to generate successively better approximations to the desired square root. To work this out, let us return to the "square root of 36 problem". As our initial guess *g1* we chose 9. We then proceeded to average this guess with its complementary value $(36/9 = 4)$. In the general case, the complementary value is given by

$$\text{complementary value} := \frac{m}{g1}$$

Our next step was to get an improved estimate of the square root, *g2*, by averaging *g1* and its complementary value (i.e. $(9+36/9)/2 = 6.5$). We can therefore write the expression for *g2* in the general case as

$$g2 := (g1 + (m/g1))/2$$

Algorithm description

1. Establish *m* the number whose square root is required and the termination condition error *e*.
2. Set the initial guess *g2* to *m*/2.
3. Repeatedly
   (a) let *g1* assume the role of *g2*,

   (b) generate a better estimate *g2* of the square root using the averaging formula,

   until the absolute difference between *g1* and *g2* is less than *error e*.
4. Return the estimated square root *g2*.

Algorithm 3.2
THE SMALLEST DIVISOR OF AN INTEGER

Problem

Given an integer n devise an algorithm that will find its smallest exact divisor other than one.

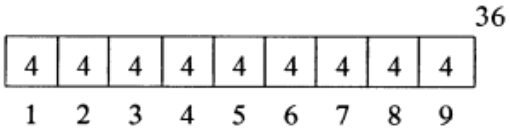
Algorithm development

Taken at face value, this problem seems to be rather trivial. We can take the set of numbers 2, 3, 4, ..., n and divide each one in turn into n. As soon as we encounter a number in the set that *exactly* divides into n our algorithm can terminate as we must have found the smallest exact divisor of n. This is all very straightforward. The question that remains, however, is can we design a more efficient algorithm?

As a starting point for this investigation, let us work out and examine the complete set of divisors for some particular number. Choosing the number 36 as our example, we find that its complete set of divisors is

$$\{2, 3, 4, 6, 9, 12, 18\}$$

We know that an *exact divisor* of a number divides into that number leaving no remainder. For the exact divisor 4 of 36, we have:



That is, there are exactly 9 fours in 36. It also follows that the bigger number 9 also divides *exactly* into 36. That is,

$$\frac{36}{4} \rightarrow 9$$

$$\frac{36}{9} \rightarrow 4$$

$$\text{and } \frac{36}{4\times 9} \rightarrow 1$$

Similarly, if we choose the divisor 3, we find that it tells us that there is a bigger number 12 that is also an exact divisor of 36. From this discussion we can draw the conclusion that exact divisors of a number must be paired.

Clearly, in this example we would not have to consider either 9 or 12 as potential candidates for being the smallest divisor because both are linked with another smaller divisor. For our complete set of divisors of 36, we see that:

Smaller factor		*Bigger factor*	
2	is linked with	18	(i.e. $\frac{36}{2} \rightarrow 18$)
3	is linked with	12	
4	is linked with	9	
6	is linked with	6	

From this set, we can see that the smallest divisor (2) is linked with the largest divisor (18), the second smallest divisor (3) is linked with the second biggest divisor (12) and so on. Following this line of reasoning through we can see that our algorithm can safely terminate when we have a pair of factors that correspond to

(a) the biggest smaller factor s,
(b) the smallest bigger factor b.

Algorithm description

1. Establish n the integer whose smallest divisor is required.
2. If n is not odd then return 2 as the smallest divisor
   else
   (a) compute r the square root of n,
   (b) initialize divisor d to 3,
   (c) while not an exact divisor and square root limit not reached do
   (c.1) generate next member in odd sequence d,
   (d) if current odd value d is an exact divisor
   then return it as the exact divisor of n
   else return 1 as the smallest divisor of n.

Algorithm 3.3
THE GREATEST COMMON DIVISOR OF TWO INTEGERS

Problem

Given two positive non-zero integers n and m design an algorithm for finding their greatest common divisor (usually abbreviated as gcd).

Algorithm development

When initially confronted with this problem, we see that it is somewhat different from other problems we have probably encountered. The difficult aspect of the problem involves the relationship between the divisors of two numbers. Our first step might therefore be to break the problem down and find all the divisors of the two integers n and m independently. Once we have these two lists of divisors we soon realize that what we must do is select the largest element common to both lists. This element must be the greatest *common* divisor for the two integers n and m.

For our example, we have:

(a) the greatest divisor of 30 is 30;
(b) the greatest divisor of 18 is 18.

Our problem is to find the greatest *common* divisor of *two* numbers rather than one number. Clearly no number greater than 18 can be a candidate for the gcd because it will not divide exactly into 18. We can in fact generalize this statement to say that the gcd of two numbers cannot be bigger than the smaller of the two numbers. The next question we might ask is, can the gcd of two numbers be *equal to* the smaller of those two numbers (this is not true in the case of 18 and 30 but if we were considering 12 and 36 we would find that 12 is the gcd)? We can therefore conclude that the smaller of the two numbers n and m must be the upper limit for the gcd.

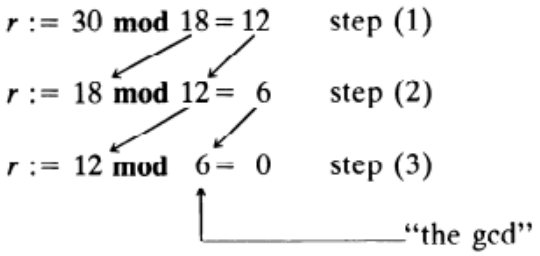
We must now decide how to continue when the smaller of the two numbers n and m is not their gcd.

Our task now is to work out the details for implementing and terminating the gcd mechanism. First let us consider how to establish if the smaller number exactly divides into the larger number. Exact division can be detected by there being no remainder after integer division. The **mod** function allows us to compute the remainder resulting from an integer division. We can use:

$$r := n \textbf{ mod } m$$

provided we had initially ensured that $n \geq m$. If r is zero, then m is the gcd. If r is not zero, then as it happens it corresponds to the "non-common" part between n and m. (E.g. 30 **mod** 18 = 12.) It is therefore our good fortune that the **mod** function gives us just the part of n we need for solving the new smaller gcd problem. Furthermore, r by definition must be smaller than m. What we need to do now is set up our iterative construct using the **mod** function. To try to formulate this construct, let us return to our gcd (18, 30) problem.

For our specific example we have:



Our example suggests that with each reduction in the problem size the smaller integer assumes the role of the larger integer and the remainder assumes the role of the smaller integer.

Algorithm description

1. Establish the two positive non-zero integers smaller and larger whose gcd is being sought.
2. Repeatedly
   (a) get the remainder from dividing the larger integer by the smaller integer;
   (b) let the smaller integer assume the role of the larger integer;
   (c) let the remainder assume the role of the divisor
   until a zero remainder is obtained.
3. Return the gcd of the original pair of integers.

Algorithm 3.4
GENERATING PRIME NUMBERS

Problem

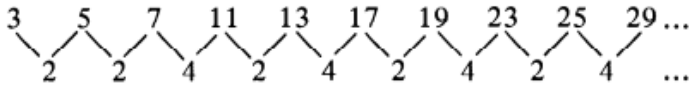
Design an algorithm to establish all the primes in the first n positive integers.

Algorithm development

The efficient generation of prime numbers is an open problem. We will consider here the more restricted problem of generating all primes in the first n integers. *A prime number is a positive integer that is exactly divisible only by 1 and itself.* The first few primes are:

2 3 5 7 11 13 17 19 23 29 31 37 ...

For large n this still leaves us with a large set of numbers to consider. So far we have eliminated numbers divisible by 2. Can we extend this to eliminating numbers divisible by 3, 5, and so on? To explore this idea let us first write down the odd sequence with the multiples of 3 removed. We have:



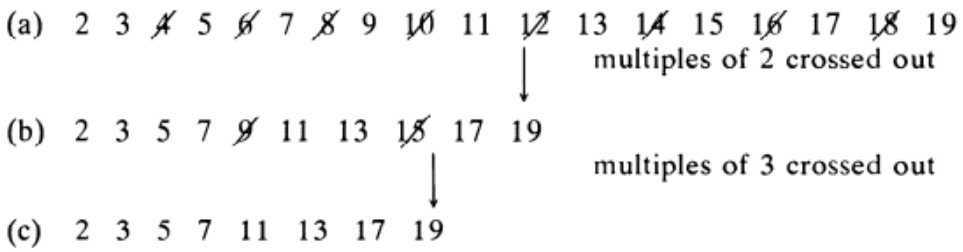
Beyond 5 we have the alternating sequence of differences 2, 4. This alternating difference sequence should be able to be generated. We will not dwell on it here but it is easy to see that the construct below with dx initially 4

$$dx := abs(dx-6)$$

has the desired behavior. This device will allow us to eliminate two-thirds of the numbers from consideration as potential prime numbers candidates.

We might now ask can we eliminate multiples of 5 in a similar manner?

The answer is yes but it would be slightly more involved. This line of attack does not seem as though it is going to be very fruitful. What we do see from this however is that one way to generate prime numbers is to simply write down the list of all integers then cross out multiples of 2, 3, 5, 7, 11, and so on.



At this stage we can propose a basic structure for our algorithm:

while $x<n$ **do**
begin
(a) generate next x using the construct $dx := abs(dx-6)$,
(b) test whether x is prime using all primes $\leqslant\sqrt{x}$,
(c) if a prime is found that is less than $\sqrt{n}$ then store it for later testing against larger x values.
end

To test all integers up to n for primality we will need to retain all primes up to $\sqrt{n}$.

Every time a new x is brought up for testing we will need to ensure that we have the appropriate set of primes to divide into x.

Working through some examples we find:

x range	*prime divisors required*
$2\leqslant x<9$	2
$9\leqslant x<25$	2, 3
$25\leqslant x<49$	2, 3, 5
$49\leqslant x<121$	2, 3, 5, 7
⋮ ⋮	⋮ ⋮

Some thought reveals there are two conditions under which this loop should terminate:

1. an exact divisor of x has been found—so it cannot be prime;
2. we have reached the divisor with index one less than limit.

Using the **mod** function to test for exact division and using the remainder *rem* to set the Boolean condition *prime* we get:

```
j := 3; prime := true;
while prime and (j<limit) do
  begin
    rem := x mod p[j];
    prime := rem <> 0;
    j := j+1
  end
```

We can use this information to "*cross out*" in advance the next value of x divisible by $p[k]$. To do this another array $out[1..\sqrt{n}]$ will be needed to store values crossed out in advance. The crossing out is done by

```
nxtout := p[k]- rem;
out[nxtout] := false
```

The idea will then be to check the array *out before* testing a given number for primality. If it is already "crossed out" no prime testing need be done. An investigation and testing of this idea shows that it will allow us to cut down by a factor of 4 or 5 the number of numbers that have to be tested for primality.

This sounds like a useful refinement. Unfortunately whenever a prime x is encountered *all* prime divisors less than $\sqrt{x}$ must be tested against it. For large n establishing the primes by this method is going to be computationally costly.

Algorithm description

1. Initialize and write out the first 3 primes. Also initialize the square of the 3rd prime.
2. Initialize x to 5.
3. While x less than n do
   (a) get next x value excluding multiples of 2 and 3;
   (b) if not past end of multiples list then
      (b.1) if $x \geq$ square of largest prime then
         (1.a) include next prime multiple as its square,
         (1.b) update square by squaring next prime $> \sqrt{x}$;
   (c) while have not established x is non-prime with valid prime multiples do
      (c.1) while current prime multiple is less than x, increment by current prime value doubled,
      (c.2) do prime test by comparing x with current multiple;
   (d) if current x prime then
      (d.1) write out x and if it is less than $\sqrt{n}$ store it.

Algorithm 3.5
COMPUTING THE PRIME FACTORS OF AN INTEGER

Problem

Every integer can be expressed as a product of prime numbers. Design an algorithm to compute all the prime factors of an integer n.

Algorithm development

Examination of our problem statement suggests that

$$n = f_1 \times f_2 \times f_3 \cdots \times f_k \text{ where } n > 1 \text{ and } f_1 \leqslant f_2 \leqslant \cdots \leqslant f_k$$

The elements $f_1, f_2, \ldots, f_k$ are all prime numbers. Applying this definition to some specific examples we get:

$$\begin{aligned} 8 &= 2 \times 2 \times 2 \\ 12 &= 2 \times 2 \times 3 \\ 18 &= 2 \times 3 \times 3 \\ 20 &= 2 \times 2 \times 5 \\ 60 &= 2 \times 2 \times 3 \times 5 \end{aligned}$$

A better and more economical strategy is therefore to only compute prime divisors as they are needed. For this purpose we can include a modified version of the sieve of Eratosthenes that we developed earlier. As in our earlier algorithm as soon as we have discovered n is prime we can terminate. At this stage let us review the progress we have made. The top-level description of the central part of our algorithm is:

while "it has not been established that n is prime" **do**
begin
(a) if *nxtprime* is divisor of n then save *nxtprime* as a factor and reduce n by *nxtprime*
else get next prime,
(b) try *nxtprime* as a divisor of n.
end

We now must work out how the "not prime" test for our outer loop should be implemented. The technique we employed earlier was to use integer division and test for zero remainder. Once again this idea is applicable. We also know that as soon the prime divisor we are using in our test becomes greater than $\sqrt{n}$ the process can terminate.

Initially when the prime divisors we are using are much less than $\sqrt{n}$ we know that the testing must continue. In carrying out this process we want to avoid having to calculate the square root of n repeatedly. Each time we make the division:

$$n \textbf{ div } nxtprime \qquad \text{(e.g. 60 } \textbf{div}\text{ 2)}$$

we know the process must continue until the quotient q resulting from this division is less than *nxtprime*.

At this point we will have:

$$(nxtprime)^2 > n$$

which will indicate that n is prime. The conditions for it not yet being established that n is prime are therefore:

(a) exact division (i.e. $r := n$ **div** $nxtprime = 0$),
(b) quotient greater than divisor (i.e. $q := n$ **mod** $nxtprime > nxtprime$).

The truth of either condition is sufficient to require that the test be repeated again.

Algorithm description

1. Establish n the number whose prime factors are sought.
2. Compute the remainder r and quotient q for the first prime $nxtprime = 2$.
3. While it is not established that n is prime do
   (a) if *nxtprime* is an exact divisor of n then
      (a.1) save *nxtprime* as a factor f,
      (a.2) reduce n by *nxtprime*,
   else
      (a'.1) get next biggest prime from sieve of Eratosthenes,
   (b) compute next quotient q and remainder r for current value of n and current prime divisor *nxtprime*.
4. If n is greater than 1 then
   add n to list as a prime factor f.
5. Return the prime factors f of the original number n.

Arrays

2.1 DEFINITION

An *array* is a finite, ordered and collection of homogeneous data elements. Array is finite because it contains only limited number of elements; and ordered, as all the elements are stored one by one in contiguous locations of computer memory in a linear ordered fashion. All the elements of an array are of the same data type (say, integer) only and hence it is termed as collection of homogeneous elements. Following are some examples:

1. An array of integers to store the age of all students in a class.
2. An array of strings (of characters) to store the name of all villagers in a village.

2.2 TERMINOLOGY

Size. Number of elements in an array is called the *size* of the array. It is also alternatively termed as *length* or *dimension*.

Type. *Type* of an array represents the kind of data type it is meant for. For example, array of integers, array of character strings, etc.

Base. *Base* of an array is the address of memory location where the first element in the array is located. For example, 453 is the base address of the array as mentioned in Figure 2.1.

Index. All the elements in an array can be referenced by a subscript like A_i or A[*i*], this subscript is known as *index*. Index is always an integer value. As each array elements is identified by a subscript or index that is why an array element is also termed as *subscripted* or *indexed variable*.

2.3 ONE-DIMENSIONAL ARRAY

If only one subscript/index is required to reference all the elements in an array then the array will be termed as *one-dimensional* array or simply an array.

2.3.1 Memory Allocation for an Array

Memory representation of an array is very simple. Suppose, an array A[100] is to be stored in a memory as in Figure 2.2. Let the memory location where the first element can be stored is *M*. If each element requires one word then the location for any element say A[*i*] in the array can be obtained as:

$$\text{Address (A}[i]) = M + (i - 1)$$

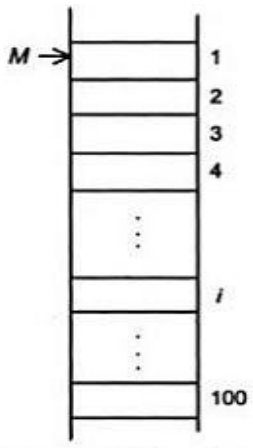


Fig. 2.2 Physical representation of a one-dimensional array.

Likewise, in general, an array can be written as A[L ... U], where L and U denote the lower and upper bounds for index. If it is stored starting from memory location M, and for each element it requires w number of words, then the address for A[i] will be

$$\text{Address (A}[i]) = M + (i - L) \times w$$

The above formula is known as *indexing formula*; which is used to map the logical presentation of an array to physical presentation. By knowing the starting address of an array M, the location of i-th element can be calculated instead of moving towards i from M. Figure 2.3.

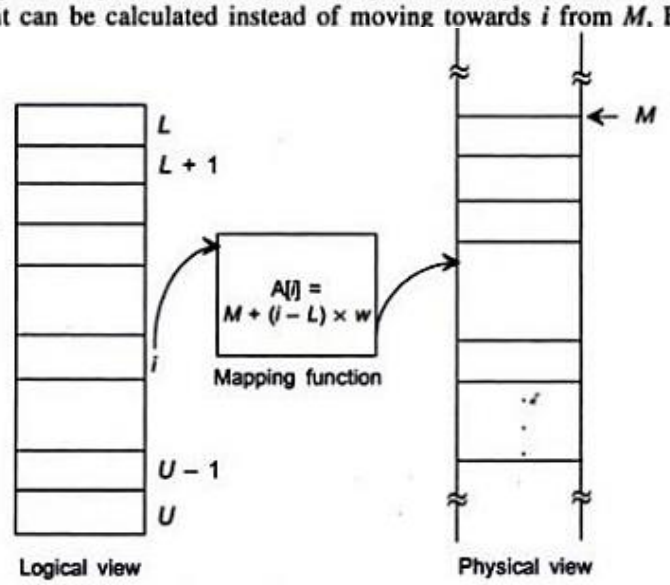


Fig. 2.3 Address mapping between logical and physical views of an array.

2.3.2 Operations on Arrays

Various operations that can be performed on an array are: traversing, sorting, searching, insertion, deletion, merging.

Traversing

This operation is used visiting all elements in an array. A simplified algorithm is presented as below:

Algorithm TRAVERSE_ARRAY( )
Input: An array A with elements.
Output: According to PROCESS( ).
Data structures: Array A[L ... U]. //L and U are the lower and upper bound //of array index

Steps:
1. $i = L$ //Start from first location L
2. While $i \leq U$ do
   1. PROCESS(A[i])
   2. $i = i + 1$ //Move to the next location
3. EndWhile
4. Stop

2.4 MULTIDIMENSIONAL ARRAYS

So far we have discussed the one dimensional arrays. But multidimensional arrays are also important. Matrix (2- dimensional array), 3-dimensional array are two examples of multidimensional arrays. The following sections describe the multidimensional arrays.

2.4.1 Two-dimensional Arrays

Two-dimensional arrays (alternatively termed as matrices) are the collection of homogeneous elements where the elements are ordered in a number of rows and columns. An example of an $m \times n$ matrix where m denotes number of rows and n denotes number of columns is as follows:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & a_{m3} & a_{m4} & \cdots & a_{mn} \end{bmatrix}_{m \times n}$$

The subscripts of any arbitrary element, say (a_{ij}) represent the i-th row and j-th column.

Memory representation of a matrix

Like one-dimensional array, matrices are also stored in contagious memory locations. There are two conventions of storing any matrix in memory:

1. Row-major order
2. Column-major order.

In row-major order, elements of a matrix are stored on a row-by-row basis, that is, all the elements in first row, then in second row and so on. On the other hand, in column-major order, elements are stored column-by-column, that is, all the elements in first column are stored in their order of rows, then in second column, third column and so on. For example, consider a matrix A of order 3 × 4:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix}_{3 \times 4}$$

This matrix can be represented in memory as shown in Figure 2.9.

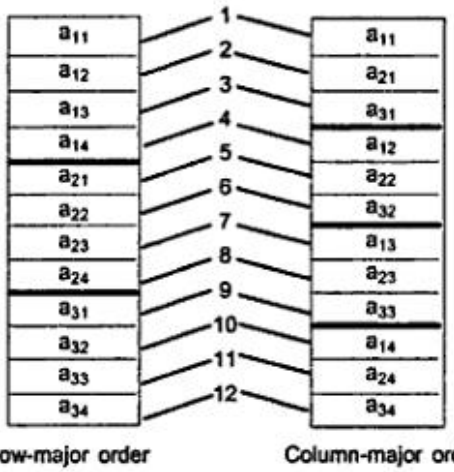


Fig. 2.9 Memory representation of $A_{3\times4}$ matrix.

Algorithm 4.1
ARRAY ORDER REVERSAL

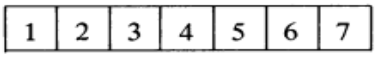
Problem

Rearrange the elements in an array so that they appear in reverse order.

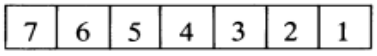
Algorithm development

The problem of reversing the order of an array of numbers appears to be completely straightforward. Whilst this is essentially true, some care and thought must go into implementing the algorithm.

We can start the design of this algorithm by careful examination of the elements of an array before and after it has been reversed; for example,

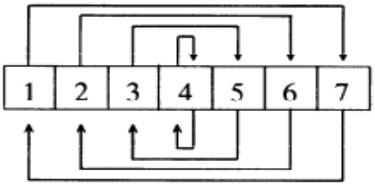


before reversal



after reversal

What we observe from our diagram is that the *first* element ends up in the *last* position. The *second* element ends up in the *second last* position and so on. Carrying this process through we get the following set of exchanges:



array $a[1..n]$

In terms of suffixes the exchanges are:

step [1]	$a[1]<=>a[7]$	
step [2]	$a[2]<=>a[6]$	
step [3]	$a[3]<=>a[5]$	
step [4]	$a[4]<=>a[4]$	there is no exchange here

Each exchange (cf. algorithm 2.1) can be achieved by a mechanism of the form

```
t := a[1];
a[i] := a[n-i+1];
a[n-i+1] := t
```

The only other aspect of the algorithm that we need to consider is the range that i can assume for a given n. Studying our original array it is clear that only 3 exchanges are needed to reverse arrays with either 6 or 7 elements. Consideration of further examples leads us to the generalization that the number of exchanges r to reverse the order of an array is always the nearest integer that is less than or equal half the magnitude of n. Our algorithm now follows directly from the above results and discussion.

Algorithm description

1. Establish the array $a[1..n]$ of n elements to be reversed.
2. Compute r the number of exchanges needed to reverse the array.
3. While there are still pairs of array elements to be exchanged
   (a) exchange the i^{th} element with the $[n-i+1]^{th}$ element.
4. Return the reversed array.

The algorithm can be suitably implemented as a procedure that accepts as input the array to be reversed and returns as output the reversed array.

Algorithm 4.2
ARRAY COUNTING OR HISTOGRAMMING

Problem

Given a set of n students' examination marks (in the range 0 to 100) make a count of the number of students that obtained each possible mark.

Algorithm development

This problem embodies the same principle as algorithm 2.2 where we had to make a count of the number of students that passed an examination. What we are required to do in this case is obtain the distribution of a set of marks. This problem is typical of frequency counting problems. One approach we could take is to set up 101 variables $C0$ $C1$, $C2$, ..., $C100$ each corresponding to a particular mark. The counting strategy we could then employ might be as follows:

while less than n marks have been examined do
(a) get next mark m,
(b0) if $m = 0$ then $C0 := C0+1$;
(b1) if $m = 1$ then $C1 := C1+1$;
(b2) if $m = 2$ then $C2 := C2+1$;
(b3) if $m = 3$ then $C3 := C3+1$;
⋮
(b100) if $m - 100$ then $C100 := C100+1$.

What we must now consider is just how the count for each array location is achieved. In setting up our mechanism we want to try to incorporate the one-step procedure that was possible in the hand solution. Initially we can consider what happens when a particular mark is encountered. Suppose the current mark to be counted is 57. In using the array for counting we must at this stage add one to the count stored in location 57. For this step we can use the actual mark's value (i.e. 57) to reference the array location that we wish to update. That is, the mark's value can be employed as an array suffix. Because it is necessary to add one to the previous count in location 57, we will need a statement of the form:

new count in location 57 := *previous count in location* 57+1

Since location $a[57]$ must play both the "previous count" and "new count" roles, we can write

$$a[57] := a[57]+1$$

or for the general mark m we can write

$$a[m] := a[m]+1$$

This last statement can form the basis of our marks-counting algorithm. By using the mark value to address the appropriate array location, we have modelled the direct update method of the hand solution.

Algorithm description

1. Prompt and read in n the number of marks to be processed.
2. Initialize all elements of the counting array $a[0..100]$ to zero.
3. While there are still marks to be processed, repeatedly do
   (a) read next mark m,
   (b) add one to the count in location m in the counting array.
4. Write out the marks frequency count distribution.

Algorithm 4.3
FINDING THE MAXIMUM NUMBER IN A SET

Problem

Find the maximum number in a set of n numbers.

Algorithm development

Before we begin to work on the algorithm for finding the maximum we need to have a clear idea of the definition of a maximum. After consideration we can conclude that the maximum is that number which is greater than or equal to all other numbers in the set. This definition accommodates the fact that the maximum may not be unique. It also implies that the maximum is only defined for sets of one or more elements.

To start on the algorithm development for this problem let us examine a particular set of numbers. For example,

8	6	5	15	7	19	21	6	13

After studying this example we can conclude that *all* numbers need to be examined to establish the maximum. A second conclusion is that comparison of the relative magnitude of numbers must be made.

Imagine for a moment that we are given the task of finding the maximum among one thousand numbers by having them flashed up on a screen one at a time. This task is close to the problem that must be solved to implement the computer algorithm. When the *first* number appears on the screen we have no way of knowing whether or not it is the maximum. In this situation the best that we can do is write it down as our temporary candidate for the maximum. Having made the decision to write down the first number we must now decide what to do when the second number appears on the screen. Three situations are possible:

1. the second number can be *less* than our temporary candidate for the maximum;
2. the second number can be *equal* to our temporary candidate for the maximum;
3. the second number can be *greater* than our temporary candidate for the maximum.

If situations (1) or (2) apply our temporary candidate for the maximum is still valid and so there is no need to change it. In these circumstances we can simply go ahead and compare the third number with our temporary maximum which we will call *max*.

Algorithm description

1. Establish an array $a[1..n]$ of n elements where $n \geq 1$.
2. Set temporary maximum *max* to first array element.
3. While less than n array elements have been considered do
   (a) if next element greater than current maximum *max* then assign it to *max*.
4. Return maximum *max* for the array of n elements.

Algorithm 4.6
FINDING THE k^{th} SMALLEST ELEMENT

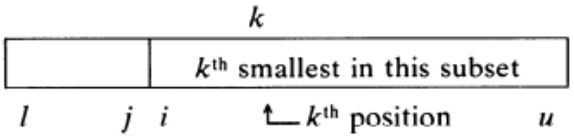
Problem

Given a randomly ordered array of n elements determine the k^{th} smallest element in the set.

Algorithm development

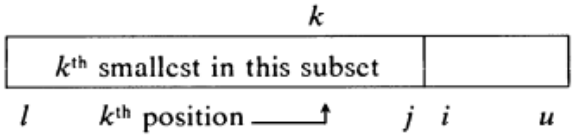
We have already seen an algorithm for finding the largest value in a set. The current problem is a generalization of this problem. As we saw in the partitioning problem (algorithm 4.5), one way to find the k^{th} smallest element would be to simply sort the elements and then pick the k^{th} value. We may, however, suspect that this problem could be treated in a similar fashion to the partitioning problem. In the partitioning problem, we knew in advance the value x about which the array was to be partitioned but we did not know how many values were to be partitioned on either side of x. The current problem represents the complementary situation where we are given how the array is to be partitioned but we do not know in advance the value of x (i.e. the k^{th} smallest value).

1. k^{th} smallest in subset $\geqslant x$:



Here we set $l := i$ and repeat the partitioning process for the new limits of l and u.

2. k^{th} smallest in subset $\leqslant x$:



Here we set $u := j$ and repeat the partitioning process for the new limits of u and l. By examining the values of i and j on termination of the partitioning loop we know which subset contains the k^{th} smallest value and hence which limit must be updated. The two tests we can use are

if $j<k$ **then** $l := i$;
if $i>k$ **then** $u := j$

The partitioning process need only continue while $l<u$. The variables i and j will need to be reset to the adjusted limits l and u before beginning each new partitioning phase.

The basic mechanism may therefore take the form:

while $l<u$ do
(a) choose some value x about which to partition the array,
(b) partition the array into two partitions marked by i and j,
(c) update limits using the tests
if $j<k$ **then** $l := i$
if $i>k$ **then** $u := j$

lem. The difference we may anticipate at this stage is that x will be selected using

$$x :- a[k]$$

rather than having x with a given value as in the earlier problem. Unlike in the partitioning problem, because x is selected from the array, we should be able to avoid the complications caused by the fact that it could be outside the bounds of the array. In the "moving-inwards" process the loop

while $a[j]>x$ **do** $j := j-1$

is guaranteed to stop because it must eventually run into x. There is, however, a problem with

while $a[i]=<x$ **do** $i := i+1$

because it will not stop on encountering the array value equal to x. Can we prevent this? The answer is yes, if we change the $\leqslant$ to a $<$ sign, for example,

while $a[i]<x$ **do** $i := i+1$

We are better off this time than with the earlier algorithm because it is no longer possible for the two **while**-loops to cause i and j to cross over *before* the last exchange in the current iteration of the outer partition loop. This is because x is always kept *between* the i and j positions. The conditional test controlling exchanges:

if $i\leqslant j$ **then** "exchange"

becomes unnecessary. We now, therefore, have a simpler and more efficient algorithm. The detailed description of our final algorithm is given below.

Algorithm description

1. Establish $a[1..n]$ and the requirement that the k^{th} smallest element is sought.
2. While the left and right partitions do not overlap do
   (a) choose $a[k]$ as the current partitioning value x;
   (b) set i to the upper limit l of the left partition;
   (c) set j to the lower limit u of the right partition;
   (d) while i has not advanced beyond k and j is greater than or equal to k do
      (d.1) extend the left partition while $a[i]<x$;
      (d.2) extend the right partition while $x<a[j]$;
      (d.3) exchange $a[i]$ with $a[j]$;
      (d.4) extend i by 1 and reduce j by 1;
   (e) if k^{th} smallest in left partition, update upper limit u of left partition;
   (f) if k^{th} smallest in right partition, update lower limit l of right partition.
3. Return the partitioned array with elements $\leqslant a[k]$ in the first k positions in the array.