"Civilization progresses by extending the number of operations that we can perform without thinking about them"

(Alfred North Whitehead, *Introduction to Mathematics*, 1911)

(This chapter is work in progress.) In previous chapters of this book we described and built the hardware architecture of a computer platform, called *Hack*, and the software hierarchy that makes it usable. In particular, we introduced an object-based language, called *Jack*, and described how to write a compiler for it. Other high-level programming languages can be specified on top of the Hack platform, each requiring its own compiler.

The last major interface which is missing in this puzzle is an *operating system*. The OS is designed to close gaps between the computer's software and hardware systems, and to make the overall computer more accessible to programmers and users. For example, our computer is equipped with a bitmap screen. In order to output the text "`Hello World`", several hundreds pixels must be drawn at specific locations on the computer's screen. To do so, we can consult the hardware specification, and write commands that put the necessary bits in the RAM segment that controls the screen's output. Needless to say, high-level programmers will need a better interface with the screen. They will want to use commands like `print('Hello World')`, and let someone else worry about the details. And that's where the operating system enters the picture.

Throughout the chapter, the term "operating system" is used rather loosely. In fact, the OS services that we describe comprise an operating system in a very minimal fashion, aiming to encapsulate various hardware-provided services in a software-friendly way and extending high-level languages with some mathematical functions and abstract data types. The dividing line between an operating system in this sense and a "standard language library" is not very clear. Usually, standard libraries associated with particular programming languages include both interfaces to the underlying services of the operating system and other libraries and services related to the programming language and its indented uses.

Indeed, the simple operating system we build here may alternatively be viewed as a standard library for the Jack language. It is packaged a collection of Jack classes, each providing a set of related services via Jack subroutine calls. The resulting OS has many features resembling those of industrial strength operating systems, but it lacks numerous OS features such as process handling or disk management.

Operating systems are usually programmed in a high level language and compiled into binary form like any other program. Indeed the operating system described here is written completely in Jack. Unlike normal programs written in a high-level language, the operating system code must be aware of the hardware platform it is running on. For example, in order to implement the various encapsulated I/O services, it must directly access the I/O devices. The Jack programming language was defined with sufficient "lowness" in it, permitting an intimate closeness to the hardware, when needed.

---

[1] From *The Elements of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2004, www.idc.ac.il/csd

The title of this chapter is somewhat misleading, since we discuss only the OS elements needed for our computer platform. The chapter starts with a background section that describes the underlying algorithms and programming techniques. Next, we specify the complete Sack OS API, and give guidelines on how to implement it in Jack. Section 4 mentions briefly some of the elements of normal operating systems that were not discussed, and Section 5 walks you through a complete implementation of the OS.

The chapter embeds two key lessons, one in software engineering and one in computer science. First, we describe and illustrate the important interplay between high-level languages, compilers, and operating systems. Second, we present a series of elegant and efficient algorithms, each being a little computer science gem.

1. Background

1.1 Mathematical Operations

Almost every computer system must support mathematical operations like addition, multiplication, and division. Normally, *addition* is implemented in hardware, at the ALU level, as we have done in Chapter 3. Other operations like *multiplication* and *division* can be implemented in either hardware or software, depending on the computer's purpose and cost/performance requirements. This section shows how multiplication, division, and square root operations can be implemented efficiently in software, at the operating system level. It should be noted that hardware implementation of these operations are based on the same algorithms presented below.

A word about algorithmic efficiency is in order here. Mathematical algorithms operate on n-bit binary numbers, with typical computer architectures having n=16 (as in Hack), 32 or 64. As a rule, we seek algorithms whose running time is proportional (or at least "polynomial") in this parameter n. Algorithms whose running time is proportional to the *value* of n-bit numbers are unacceptable, since these values are exponential in n. For example, suppose we implement the multiplication operation $x \cdot y$ using the "repeated addition" algorithm *for i* = 1 ... *y* {*sum* = *sum* + *x*}. If we use this algorithm on a 64-bit computer, y can be as large as 1,000,000,000,000,000,000 (still smaller than the maximal value 2^{63}). In such cases, this naïve algorithm may run for years, even on the fastest computers. On the other hand, the running time of the multiplication algorithm that we present below is proportional to the number of bits n, and thus will require only dozens or a few hundreds of operations on a 64-bit architecture for any value of y.

We will use the standard "Big-Oh" notation, $O(n)$, to describe the running time of algorithms. Readers who are not familiar with this notation should simply read $O(n)$ as "in the order of magnitude of n". With that in mind, we now turn to present a multiplication $x \cdot y$ algorithm for n-bit numbers whose running time is $O(n)$ rather than $O(y)$, which is exponentially larger.

Multiplication

Consider the standard multiplication method taught in elementary school. To compute 356 times 27, we write the two numbers one on top of the other. Next, we multiply each digit of 356 by 7. Next, we "shift to the left" one position, and multiply each digit of 356 by 2. Finally, we sum up the numbers in the two rows and obtain the result. The binary version of this technique -- Algorithm 1 -- is exactly the same.

The "steps"

```
    1 0 1 1  =  1 1
      1 0 1  =    5
  ---------
    1 0 1 1
  0 0 0 0
1 0 1 1
---------------
1 1 0 1 1 1  =  5 5
```

The algorithm explained (first 4 of 16 iteration)

(ignoring some leading zeros, to save clutter)

x:	0	0	0	1	0	1	1	
y:	0	0	0	0	1	0	1	j'th bit of y
	0	0	0	**1**	**0**	**1**	**1**	1
	0	0	**1**	**0**	**1**	**1**	0	0
	0	**1**	**0**	**1**	**1**	0	0	1
	1	**0**	**1**	**1**	0	0	0	0
$x \cdot y$:	0	1	1	0	1	1	1	sum

```
multiply(x,y):
  initialize sum = 0
  initialize shiftedX = x
  for j = 0…(n−1) do
    if (j'th bit of y) = 1 then
      sum = sum + shiftedX
    shiftedX = shiftedX * 2
```

ALGORITHM 1: Multiplication.

This algorithm performs $O(n)$ addition operations on n-bit numbers, where n is the number of bits in x and y. (Note that *shiftedX* * 2 can be efficiently obtained by either adding *shiftedX* to itself or shifting its bit representation one place to the left.)

Division

The naïve way to compute x / y is to repeatedly subtract y from x until it is impossible to continue (i.e. until $x<y$). The running time of this algorithm is clearly proportional to the quotient, and may be as large as $O(x)$, which is exponential in the number of bits n. To speed up this algorithm, we can try to subtract large chunks of y's from x in each iteration. For example, if x=891 and y=5, we can tell right away that we can deduct a hundred 5's from x and the remainder will still be greater than 5, thus shaving 100 iterations from the naïve approach. Indeed, this is the rationale behind the school method for long division. Formally, in each iteration we try to subtract the largest possible shift of y, i.e. $y \cdot T$ where T is the largest power of 10 such that $y \cdot T \le x$. The binary version of this algorithm is precisely the same, except that T is a power of 2 instead of 10.

It is an easy exercise to formally write down this school algorithm for division, as we have done for multiplication. We find it more illuminating to provide the same logic in the form of a recursive program that is probably easier to implement:

```
divide (x,y):
  // Integer part of x/y, where x and y are natural numbers.
  if  y>x return 0
  q = divide(x, 2*y)
  if (x – 2*q*y) < y
      return 2*q
  else
      return 2*q + 1
```

ALGORITHM 2: Division.

The running time of this algorithm is determined by the depth of the recursion. Since in each level of recursion the value of y is multiplied by 2, and since we terminate once $y>x$, it follows that the recursion depth is bounded by the number of bits in x. Each recursion level involves a constant number of addition, subtraction, and multiplication operations, and thus the total running time of the algorithm requires $O(n)$ such operations.

Algorithm 2 may be considered sub-optimal since each multiplication operation also requires $O(n)$ addition and subtraction operations. However, careful inspection reveals that the product 2*q*y can be computed without any multiplication. Instead, we can rely on the value of this product in the previous recursion level, and use a few addition operations to establish its current value.

Square Root

Square roots can be computed efficiently in a number of different ways, e.g. using the Newton-Raphson method or a Taylor series expansion. For our purposes though, a simple binary search will suffice. The square root function $y = \sqrt{x}$ has two convenient properties. First, it is monotonically increasing. Second, its inverse function $x = y^2$ is something that we already know how to compute (multiplication). Taken together, these properties imply that we have all we need to compute square roots using binary search.

```
sqrt(x):      // Compute the integer part of y = √x :
  // Find y such that y² ≤ x < (y+1)² :
  initialize low = 0
  initialize high = square root of the largest n-bit number
  while low < high do
        med = (low + high) / 2
        if med * med > x
            high = med - 1
        else
            low = med
  return low
```

ALGORITHM 3: Square root computation using binary search.

Note that each loop iteration takes a constant number of arithmetic operations. Since the difference between *high* and *low* shrinks by a factor of 2 in each iteration, the total number of iterations is at most the logarithm of the initial value of *high*-*low*, which is at most *n*. Thus the total running time is $O(n)$ arithmetic operations.

1.2 String representation of numbers

Computers represent numbers in memory using binary codes. Yet humans are used to dealing with numbers in a decimal notation. Thus, when humans have to read or input numbers, and only then, a conversion to or from decimal notation must be performed. Typically, this service is implicit in the character handling services provided by the operating system. In other words, programmers can write high level code that operates directly on the decimal or textual representation of numbers, assuming that the OS will perform the necessary conversions, as needed. We now turn to describe how some of these OS services are actually implemented.

Of course the only subset of characters which is of interest here are the 10 digits which represent actual numbers. The ASCII codes of these characters are as follows:

Character:	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
ASCII code:	48	49	50	51	52	53	54	55	56	57

As gleaned from the table, single digit characters can be easily converted into their numeric representation, and vice versa, as follows. To compute the ASCII code of a given digit 0<=x<=9, we can simply add *x* to 48 – the code of '0'. Conversely, the numeric value represented by an

ASCII code 48<=y<=57 is obtained by *y* - 48. And once we know how to convert single digits, we can proceed to convert any given integer. These conversion algorithms can be based on either an iterative or a recursive logic, so we present one of each.

```
// Convert a number to a string.
toString(n):
    lastDigit = n % 10
    c = character representing lastDigit
    if n<10
       return c (as a string)
    else
       return toString(n/10).append(c)
```

```
// Convert a string to a number.
toInt(s):
    n =0
    for i = 1 .. length of s
        d = integer value of the digit s[i]
        n=n*10 + d
    return n
```

(Assuming that s[1] is the most significant digit character of s.)

ALGORITHMS 4-5: String-numeric conversion

1.3 Memory Management

Dynamic Memory Allocation: Some of the memory required for a program's execution is explicitly defined in the program code. For example, *static variables* are allocated when the program starts running, *local variables* are allocated when a subroutine starts running, and so on. Other memory is dynamically requested during the program's execution. For example, memory should be allocated dynamically to accommodate the construction of new objects or arrays whose size is determined only during run-time. This dynamic memory allocation is typically done by the operating system. When a running program constructs a new object of a certain size, enough RAM space must be located in memory and then allocated to store the new object. When the program declares that the object is no longer needed, its RAM space may be recycled. The RAM segment from which memory is dynamically allocated is called the *heap.*

Operating systems use various techniques for handling dynamic memory allocation and de-allocation. These techniques are implemented in two functions traditionally called `alloc()` and `deAlloc()`. We present two memory allocation algorithms: a basic one and an improved one.

Basic memory allocation algorithm: The data structure that this algorithm manages is a single pointer, called *free*, which points to the beginning of the heap segment that was not yet allocated. Algorithm 6-a gives the details.

```
// Objects and arrays are stored on the heap.
Initialization: free=heapBase
// Allocate a memory segment of size words:
alloc(size):
  pointer = free
  free += size
  return pointer
// De-allocate the memory space of a given object:
deAlloc(object):
  do nothing
```

ALGORITHM 6-a: Basic Memory Allocation Scheme (wasteful)

Algorithm 6-a is clearly wasteful, as it does not reclaim the space of decommissioned objects.

Improved memory allocation algorithm: This algorithm manages a linked list of available memory blocks, called *freeList*. Each block is characterized by two "housekeeping" fields: the block's length, and a pointer to the next block in the *freeList*. These fields can be kept in the two memory locations preceding the block itself. For example, the implementation can use the convention `b.length==x[-1]` and `b.next==x[-2]`.

When asked to allocate a memory segment of size *n*, the algorithm has to search the *freeList* for a suitable block. There are two well-known strategies for doing this. *Best-fit* finds the block whose size is the closest (from above) to the required size, while *first-fit* finds the first block that is long enough. Once the block has been found, the required memory segment is taken from it. Next, this block is updated in the *freeList,* becoming the part that remained after the allocation (if no memory was left in the block, the entire block is eliminated from the *freeList*).

When asked to reclaim the memory of an unused object, the algorithm inserts the de-allocated block into the *freeList*. The details are given in Algorithm 6-b.

```
// Objects and arrays are stored on the heap.
initialization:
    freeList = heapBase+2
    freeList.length = heapEnd-(heapBase+2)
    freeList.next = null
// Allocate a memory segment of size words:
alloc(size):
  1. use methods like best-fit or first-fit
     to locate a free block in freeList
  2. return the base address of that block
// De-allocate the memory space of a given object:
deAlloc(object):
  Append the object to the freeList
```

ALGORITHM 6-b: Improved Memory Allocation Scheme (with memory recycling)

After a while, dynamic memory allocation schemes like Algorithm 6-b may create a block fragmentation problem. Hence, some kind of "defrag" operation should be considered, i.e. merging memory segments that are physically consecutive in memory but logically split into different blocks in the *freeList*. The defragmentation operation can be done each time an object is de-allocated, or when `alloc()` cannot find an appropriate block, or according to some other intermediate or ad-hoc condition.

1.4 Variable length arrays and Strings

The memory allocation operations considered above allocate fixed length memory blocks. This is exactly appropriate for arrays in high level programming languages. Most programming languages also provide data-types that have variable length – most commonly strings. Strings contain arrays of characters, whose length may vary. String objects are usually provided by the standard library of the programming language (e.g. the `String` and `StringBuffer` classes in Java or the `strXXX` functions in C).

Indeed, the implementation of variable length strings can be done by creating a `String` class that provides the string abstraction and related services. The standard data structure used in this context typically contains an array of characters that holds the string contents, and the current length of the string. Array locations beyond the current length are not considered part of the string contents. When such a data structure is first constructed, some maximum possible length must be defined for it, and the array is allocated to be in this size.

1.5 Input/Output Management

An important part of the functionality of an operating system is handling the various I/O devices connected to the computer, encapsulating the details of interfacing them, and providing convenient access to their basic functionality. We will describe only the very basic elements of

handling the I/O devices available in Hack: a screen and a keyboard. We will divide the issue of handling the screen into two logically separate steps: handling graphics, and handling character output.

1.5.1 Graphics output

Pixel drawing: Most computers today use *raster*, also called *bitmap,* display technologies. The only primitive operation that can be physically performed in such an output device is that of drawing a single pixel (a *pixel* refers to a single "dot" on the screen). Pixels are specified using (*column*, *row*) coordinates. The usual convention is that columns are numbered from left to right (like the *x*-axis in high school) while rows are numbered from the top down (opposite of the *y*-axis in high school). Thus the top left pixel is located in screen location (0,0).

The low level drawing of a single pixel is a hardware-specific operation that depends on the particular interface of the screen or the underlying graphics card. If the screen interface is based on a memory map, then drawing a pixel is achieved by writing the proper value into the RAM location that represents the required pixel in memory.

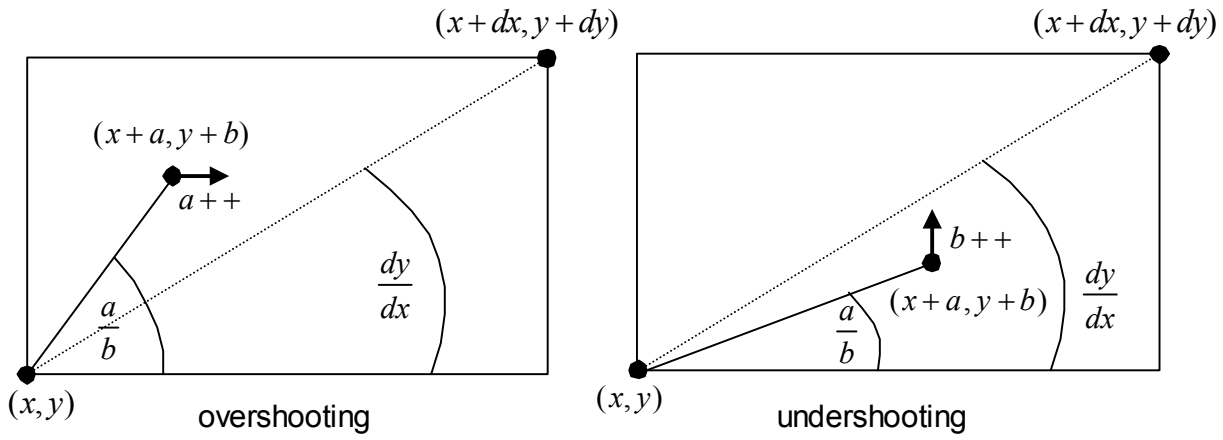
drawPixel (x,y):
// Hardware-specific. Assuming a memory mapped screen:
Write a pre-determined value in the RAM
location corresponding to screen location (x,y).

ALGORITHM 7: Drawing a pixel.

Now that we know how to draw a single pixel, we turn to describe how to draw lines and circles.

Line drawing: Recall that the only elementary drawing operation supported by computers is that of drawing a single pixel. Hence, when asked to draw a line between two screen locations, the best that we can possibly do is approximate the line by drawing a series of pixels along the imaginary line that connects the two points. Note that the "pen" that we use can move in four directions only: up, down, left, and right. Thus the drawn line is bound to be jagged, and the only way to make it look good is to use a high-resolution screen. Since the receptor cells in the human eye's retina also form a grid of "biological pixels," there is a limit to the image granularity that the human eye can resolve. Thus, high-resolution screens and printers can fool the human eye to believe that the lines drawn by pixels or printed dots are actually smooth. In fact they are always jagged.

The procedure for drawing a line from location (x1,y1) to location (x2,y2) starts by drawing the (x1,y1) pixel, and then zigzagging in the direction of the (x2,y2) pixel, until it is reached. See Algorithm 8a for the details.



drawLine(x,y,x+dx,y+dy):
// Assuming $dx, dy \geq 0$.
initialize $(a,b) = (0,0)$
while $a \leq dx$ or $b \leq dy$ do
    drawPixel$(x+a, y+b)$
    if $a/dx < b/dy$ then a++ else b++

ALGORITHM 8-a: Line Drawing

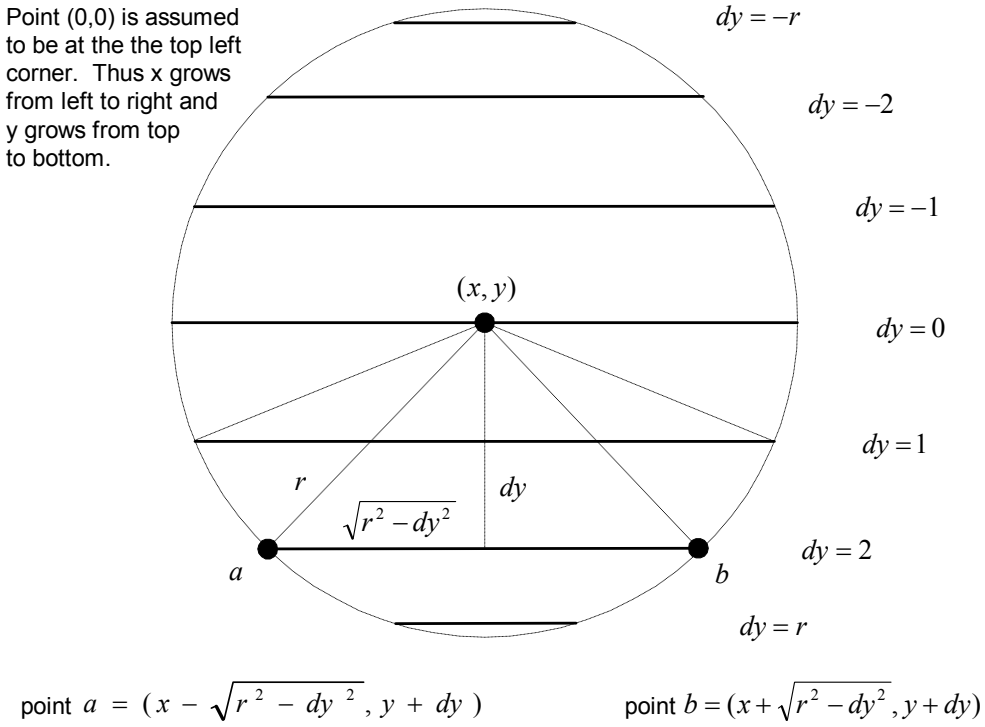
Algorithm 8-a is applicable only for $dx, dy \geq 0$. To extend it into a general-purpose line drawing routine, one also has to take care of the three other possibilities: $dx, dy \prec 0$, $dx \succ 0, dy \prec 0$, and $dx \prec 0, dy \succ 0$.

An annoying feature of this algorithm is the use of division operations (a/dx, b/dy) in each loop iteration. This division operation is not only time-consuming -- it also requires floating point operations rather than simple integer arithmetic. A possible solution is to replace the `a/dx<b/dy` condition with the equivalent `a*dy<b*dx`, which requires only integer multiplication. Further, careful inspection reveals that this latter condition may be checked without using any multiplication. As shown in Algorithm 8-b, this may be done by maintaining a variable that updates the value of `a*dy-b*dx` each time either *a* or *b* are modified.

// To test whether $a/dx < b/dy$,
// maintain a variable *adyMinusbdx*, and test whether *adyMinusbdx*<0:
Initialization: set *adyMinusbdx*=0
When a++ is performed: set adyMinusbdx = adyMinusbdx + dy
When b++ is performed: set adyMinusbdx = adyMinusbdx - dx

ALGORITHM 8-b: Testing whether $a/dx < b/dy$

Circle drawing: There are several ways to draw circles on a bitmap screen. We present an algorithm that uses three routines already implemented in this chapter: *multiplication*, *square root* computation, and *line drawing*.



drawCircle(x,y,r):

for each $dy \in -r \ldots r$ do

drawLine from $(x - \sqrt{r^2 - dy^2}, y + dy)$ to $(x + \sqrt{r^2 - dy^2}, y + dy)$

ALGORITHM 9: Circle Drawing

The algorithm is based on drawing a series of horizontal lines (like the typical line *ab* in the figure), one for each row in the range $y - r$ to $y + r$. Since *r* is specified in pixels, the algorithm ends up drawing a line in every screen row along the circle's north-south diameter, and thus the resulting circle is completely filled. A trivial version of this algorithm can yield an empty circle as well.

1.5.2 Character Output

All the output that we described so far was graphical: pixels, lines, and circles. We now turn to describe how characters are printed on the screen. Well, pixel by pixel. The first step before writing characters on a screen is to divide the physical pixel-oriented screen into a logical character-oriented screen suitable for drawing complete characters. For example, consider a

physical 256 rows by 512 columns screen. If we allocate a grid of 11*8 pixels for drawing a single character (11 rows, 8 columns), then our screen can show 23 lines, each holding 64 characters (with 3 extra rows of pixels left unused). Note that this calculation accounts for the requisite spacing, since the 11*8 allocation includes a 1-pixel space between adjacent lines and a 2-pixels space between adjacent characters.

Now, for each character that we want to display on the screen, we have to design a suitable bitmap. For example, Figure 10 gives a possible bitmap for the letter "A".

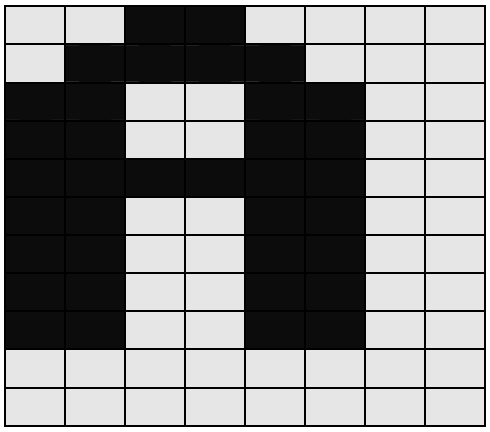


FIGURE 10: Character bitmap of the letter "a".

Characters are usually drawn on the screen one after the other, from left to right. For example, the two commands `print("a")` and `print("b")` probably mean that the programmer wants to see the image "`ab`" drawn on the screen. Thus the character-writing package must maintain a "cursor" object that keeps track of the screen location where the next character should be drawn on the logical "character screen". The cursor information consists of "line" and "column" counts. For example, the character screen described in the previous paragraph is characterized (excuse the pun) by $0 \leq \text{line} \leq 22$ and $0 \leq \text{column} \leq 63$. Drawing a single character at location (*line*, *column*) is achieved by writing the bitmap of the character onto the box of pixels at rows *line**11 … *line**11 + 10, and columns *column**8 … *column**8+7. After a character is drawn, the cursor should be moved one step to the right (i.e. *column* is increased by 1), and when a new line is requested, *row* is increased by 1 and *column* is reset to 0. When the bottom of the screen is reached, there is a question of what to do next, the common solution being to effect a "scrolling" operation. Another possibility is erasing the screen and staring over from the top left corner (i.e. setting the cursor to (0,0).)

To conclude, we know how to write characters on the screen. Writing other types of data is now easy: strings are written character by character; numbers are written by first converting them to strings, and so on.

1.5.3 Keyboard Handling

Handling user-supplied character input is more involved than meets the eye. When interacting with a computer, a human user presses a key on the keyboard for some variable duration of time. Yet the program that manages the interaction with the user wants to accept this single character

input independently of the time that elapsed between the "key press" and "key release" events.. Further, we usually want to give some feedback to the user. First, we typically want to display some graphical cursor at the screen location where the input "goes". Second, we typically want to echo the actual input by displaying it on the screen at that point.

In the "raw" form of keyboard access, the program gets direct data from the keyboard indicating which key is *currently* pressed by the user. The access to this raw data depends on the specifics of the keyboard interface. For example, if the keyboard is represented using a memory map, we can simply inspect the contents of the relevant RAM area to determine which key is presently pressed. The details of this inspection can then be incorporated into the implementation of Algorithm 11.

```
keyPressed (x,y):
// Depends on the specifics of the keyboard interface.
if a key is pressed on the keyboard
   return the ASCII value of the key
else
  return 0
```

ALGORITHM 11: Capturing ("raw") keyboard input.

Usually, an input typed by the user is considered final only after the "enter" key has been pressed, yielding the *new-line* character. Further, users may backspace and erase their previously entered characters until this event takes place. These requirements, along with the "raw" input form supplied by the `keyPressed` routine, can be used to implement the "cooked" form of character input expected by human users. The details are shown in Algorithms 12 and 13.

```
// Read and echo a single character.
readChar():
    display the cursor
    while no key is pressed on the keyboard
        do nothing  // wait till the user presses a key
    c = code of currently pressed key
    while a key is pressed
          do nothing // wait for the user to let go
    print c at the current cursor location
    move the cursor one position to the right
    return c
```

```
// Read a "line" (until new-line).
readLine():
   s = empty string
   repeat
       c = readChar()
       if c == new-line character
           print new-line
           return s
       else if c == backspace character
           remove last character from s
           move the cursor 1 pos. back
       else
           s = s.append(c)
   return s
```

ALGORITHMS 12-13: Capturing ("cooked") keyboard input.

2. The Sack OS Specification

This section duplicates "The Jack Standard Library" section from Chapter 9. The various services of the Sack operating system are organized in eight modules, as follows:

- **Math:** Provides basic mathematical operations;
- **String:** Implements the `String` type and basic string-related operations;
- **Array:** Defines the Array type and allows construction and disposal of arrays;
- **Output:** Handles text based output;
- **Screen:** Handles graphic screen output;
- **Keyboard:** Handles user input from the keyboard;
- **Memory:** Handles memory operations;
- **Sys:** Provides some execution-related services.

This section specifies the subroutines that are supposed to be in these classes.

Math

This class enables various mathematical operations.

- Function void **`init`**`()`.
- Function int **`abs`**(int x): Returns the absolute value of x.
- Function int **`multiply`**(int x, int y): Returns the product of x and y.
- Function int **`divide`**(int x, int y): Returns the integer part of the x/y.
- Function int **`min`**(int x, int y): Returns the minimum of x and y.
- Function int **`max`**(int x, int y): Returns the maximum of x and y.
- Function int **`sqrt`**(int x): Returns the integer part of the square root of x.

String

This class implements the `String` data type and various string-related operations.

- Constructor String **`new`**(int maxLength): Constructs a new empty string (of length zero) that can contain at most maxLength characters.
- Method void **`dispose`**(): Disposes this string.
- Method int **`length`**(): Returns the length of this string.
- Method char **`charAt`**(int j): Returns the character at location j of this string.
- Method void **`setCharAt`**(int j, char c): Sets the j'th element of this string to c.
- Method String **`appendChar`**(char c): Appends c to this string and returns this string.
- Method void **`eraseLastChar`**(): Erases the last character from this string.

- Method int **intValue**(): Returns the integer value of this string (or at least of the prefix until a non numeric character is found).
- Method void **setInt**(int j): Sets this string to hold a representation of j.
- Function char **backSpace**(): Returns the backspace character.
- Function char **doubleQuote**(): Returns the double quote (“) character.
- Function char **newLine**(): Returns the newline character.

Array

This class enables the construction and disposal of arrays.

- Function Array **new**(int size): Constructs a new array of the given size.
- Method void **dispose**(): Disposes this array.

Output

This class allows writing text on the screen.

- Function void **init**().
- Function void **moveCursor**(int i, int j): Moves the cursor to the j’th column of the i’th row, and erases the character located there.
- Function void **printChar**(char c): Prints c at the cursor location and advances the cursor one column forward.
- Function void **printString**(String s): Prints s starting at the cursor location, and advances the cursor appropriately.
- Function void **printInt**(int i): Prints i starting at the cursor location, and advances the cursor appropriately.
- Function void **println**(): Advances the cursor to the beginning of the next line.
- Function void **backSpace**(): Moves the cursor one column back.

Screen

This class allows drawing graphics on the screen. Column indices start at 0 and are left-to-right. Row indices start at 0 and are top-to-bottom. The screen size is hardware-dependant (over HACK: 256 rows * 512 columns).

- Function void **init**().
- Function void **clearScreen**(): Erases the entire screen.
- Function void **setColor**(boolean b): Sets the screen color (white=false, black=true) to be used for all further drawXXX commands.
- Function void **drawPixel**(int x, int y): Draws the (x,y) pixel.

- Function void **drawLine**(int x1, int y1, int x2, int y2): Draws a line from pixel (x1,y1) to pixel (x2,y2).
- Function void **drawRectangle**(int x1, int y1, int x2, int y2): Draws a filled rectangle where the top left corner is (x1, y1) and the bottom right corner is (x2,y2).
- Function void **drawCircle**(int x, int y, int r): Draws a filled circle of radius r<=181 around (x,y).

Keyboard

This class allows reading inputs from the keyboard.

- Function void **init**().
- Function char **keyPressed**(): Returns the character of the currently pressed key on the keyboard; if no key is currently pressed, returns 0.
- Function char **readChar**(): Waits until a key is pressed on the keyboard and released, and then echoes the key to the screen and returns the character of the pressed key.
- Function String **readLine**(String message): Prints the message on the screen, reads the next line (until a newline character) from the keyboard, echoes the line to the screen, and returns its value. This method handles user backspaces.
- Function int **readInt**(String message): Prints the message on the screen, reads the next line (until a newline character) from the keyboard, echoes the line to the screen, and returns the integer until the first non numeric character in the line. This method handles user backspaces.

Memory

This class allows direct access to the main memory.

- Function void **init**().
- Function int **peek**(int address): Returns the value of the main memory at this address.
- Function void **poke**(int address, int value): Sets the value of the main memory at this address to the given value.
- Function Array **alloc**(int size): Allocates the specified space on the heap and returns a reference to it.
- Function void **deAlloc**(Array o): De-allocates the given object and frees its memory space.

Sys

This class supports some execution-related services.

- Function void **init**(): Calls the `init` functions of the other OS classes (where appropriate) and then calls the `Main.main()` method.
- Function void **halt**(): Halts the program execution.

- Function void **`error`**(int errorCode): Prints the error code on the screen and halts.
- Function void **`wait`**(int duration): Waits approximately *duration* milliseconds and returns.

3. Implementation

This section provides some hints and suggestions for implementing the various classes of the Sack OS using Jack over the Hack platform.

Initialization: In each OS class that requires class-level initialization, the class-level initialization code is embedded in a single `init` routine. This routine is then called (once) by the OS's `Sys.init` routine, as part of the "booting" process. This is explained further below, in the `Sys` class implementation tips.

Math: The multiplication and division algorithms 1 and 2 are designed for natural (non-negative) numbers only. A simple way of handling negative numbers is doing all calculations on absolute values and then setting the sign appropriately. For the multiplication algorithm, this is not really needed since it turns out that if the input numbers are given in 2's complement then the results will be correct with no further ado.

Note that in each iteration j of the multiplication Algorithm 1, the j^{th} bit of the second number is extracted. We suggest to encapsulate this operation as follows:

`bit(x,j)`: Returns true if the j^{th} bit of the integer x is 1 and false otherwise.

The `bit(x,j)` function can be easily implemented using shifting operations. Alas, Jack does not support shifting. Instead, to speed up this function implementation in Jack, it may be convenient to define a fixed static array of length 16, say `twoToThe[j]`, whose j^{th} location holds the value 2 to the power of j. This array may be initialized once (in `Math.init`), and then used, via bitwise Boolean operations, in the implementation of `bit(x,j)`.

In the division Algorithm 2 we multiply y by a factor of 2 until $y>x$. A detail that needs to be taken into account is that y can overflow. This overflow can be detected by noting when y becomes negative.

For computing the square root (Algorithm 3), notice that $182^2 = 33124 > 32767 = 2^{15} - 1$, and thus the binary search can be limited to the range 0..181.

String: The ASCII codes of newline, backspace and doubleQuote are 128, 129 and 34 respectively.

Array: Note that the "`new`" function is not really a constructor, despite the fact that it looks like one. Therefore, memory space for a new array should be explicitly allocated using a call to `Memory.alloc()`. Similarly, de-allocation must be done explicitly.

Output: We suggest using character maps of 11*8, leading to 23 lines of 64 characters each. Since building character maps of all ASCII characters is quite a burden, we supply such maps for you (except for one or two characters which are left as an exercise). In particular, we supply Jack code that gives for each printable ASCII character an array that holds its bit map (using a "font" that we created). The array holds 11 entries, each corresponding to a row of pixels. Each row is given as a binary number whose bits represent the 8 pixels in the row.

There is no need to implement scrolling.

Memory: The `peek` and `poke` functions should provide direct access to the underlying memory. As it turns out, the Jack language includes a trapdoor that enables the programmer to gain complete control of the computer's memory. This hacking trick can be exploited to enable the implementation of `peek` and `poke` using plain Jack programming. The trick is based on an anomalous use of reference variables (pointers). Specifically, the Jack language does not prevent the programmer from assigning a constant to a reference variable. This constant can then be treated as an absolute memory address. In particular, when the reference variable happens to be an array, this trick can give convenient and direct access to the entire computer memory:

```
// To create a Jack-level "proxy" of the RAM:
var Array memory;
let memory=0;
// From this point on we can use code like:
let x = memory[j]  // where j is any RAM address
let memory[j] = y  // where j is any RAM address
```

PROGRAM 14: A trapdoor enabling complete control of the RAM from Jack.

Following the first two lines of Program 14, the base of the `memory` array points to the first address in the computer's RAM. To set or get the value of the RAM location whose physical address is j, all we have to do is manipulate the array entry `memory[j]`. This will cause the compiler to manipulate the RAM location whose address is `0+j`, which is precisely what we want to do.

Recall that in Jack, arrays are not allocated space on the heap at compile-time, but rather at run-time, when the array's `new` method is called. Here, however, a `new` initialization will defeat the purpose, since the whole idea is to anchor the array in a particular address rather then let the OS allocate it to an address in the heap that we don't control. In short, this hacking trick works because we use the array variable without initializing it "properly", as we would do in normal usage of arrays.

The higher level functions `alloc` and `deAlloc` manipulate the heap. Recall that the standard implementation of the VM over the Hack platform specified that the heap resides at RAM locations 2048-16383.

Screen: Drawing a pixel on the screen is done by directly accessing its memory map using `Memory.peek()` and `Memory.poke()`. Recall that the memory map of the screen on the hack

platform specifies that the pixel at column c and row r ($0 \le c \le 511$, $0 \le r \le 255$) is mapped to the c%16 bit of memory location 16384 + r*32 + c/16. Notice that drawing a single pixel requires changing a single bit in the accessed word, a task that can be achieved in Jack using bit-wise operations.

The only tricky element in the other graphic operations here is avoiding overflow. Overflow in the line drawing Algorithm 8 will not occur if you use the suggested efficient implementation for determining whether a/dx<b/dy.

The specification of the `drawCircle` routine limits circle radiuses to be at most 181. This eliminates the possibility of overflow when using the suggested circle drawing Algorithm 9.

Keyboard: Recall that the memory map of the keyboard on the Hack platform is at memory location 24576. The method `keyPressed()` provides "raw" access to this memory location and can be implemented easily using `Memory.peek()`. The other methods provide the required "cooking".

Sys: An application program written in Jack is a collection of classes. One class must be named `Main`, and this class must include a method named `main`. In order to start running the application program, the `Main.main()` method should be invoked. Now, it should be understood that the operating system is itself a program. Thus, when the computer boots up, we want to start running the operating system program first, and then we want the OS to start running the application program. Indeed, the VM specification states a bootstrap code that automatically invokes a VM function called `Sys.init()`. This `Sys.init()` function, which is part of the OS's `Sys` class, should then call the `init()` methods of the other OS classes (libraries), and then call the `Main.main()` method of the application program.

The `Sys.wait` function can be implemented pragmatically, under the limitations of the Hack platform. In particular, you can use a loop that runs approximately *n* milliseconds before it (and the method) returns. You will have to time your specific computer to obtain a one millisecond wait (this constant varies from one CPU to another). As a result, your `Sys.wait()` function will not be portable, but that's life.

The `Sys.halt` function can be implemented by entering an infinite loop.

4.Perspective

The standard class library presented in this chapter was given the name "operating system" due to its main conceptual goal: encapsulation of the gory hardware details, omissions, and idiosyncrasies in a clean software packaging. However, the gap between what we called here an operating system and an "industrial strength" operating system is rather large.

Our "operating system" completely lacks some of the very basic components most closely associated with operating systems. The Hack/Jack system supports no multi-threading or multi-processing; in contrast the very kernel of most operating systems is devoted to exactly that. The Hack/Jack system has no mass storage devices; in contrast the main information kept and handled by operating systems is the file system. The Hack/Jack system has neither a textual user interface

(as in a Unix shell or a DOS window), nor a graphical one (windows, mouse, icons, etc.); in contrast this is the operating system aspect that users expect to see and interact with. Numerous other services commonly found in operating systems are not present here: security, communication, and more.

A central feature of most operating systems is that their code is somehow more "privileged" than user code – the hardware platform forbids non-system code to perform various operation that are allowed to OS code. Consequently, access to operating system services requires a mechanism that is more elaborate than a simple function call. Further, programming languages usually wrap these OS services in regular functions or methods. In our case there is no difference between normal code and OS code, and OS system services run in the same "user mode" as that of the application program.

Our operating system does however include some of the most fundamental OS services, e.g. managing memory, driving I/O, handling initialization, as well as supplying mathematical functions not implemented in hardware. Additionally, our operating system supplies some services that are normally found in the standard libraries of programming languages, e.g. the *String* abstraction. Consistent with the spirit of this book, all these OS services are described and implemented in the simplest possible way, but not simpler.

The algorithms that we presented for multiplication and division are very standard. However, in most cases these algorithms, or a variant thereof, are implemented in hardware rather than in software. The running time of the presented multiplication and division algorithms is $O(n)$ addition operations. Since adding two n-bit numbers requires $O(n)$ bit operations (gates in hardware), multiplication and division require $O(n^2)$ bit operations. There are algorithms whose running time is asymptotically significantly faster. For a large number of bits, these algorithms are more efficient.

5. Build It

This project describes a modular implementation of the entire Sack operating system. The OS is implemented as a collection of eight classes. Each of these classes can be implemented in isolation. Further, the classes may be developed and incrementally tested in any particular order.

Objective: Implement the Sack operating system and test it by executing application programs that use OS services.

Resources: The main tool that you need in this project is Jack -- the language in which you have to develop the OS. This implies that you will also need the supplied *Jack compiler*, to compile your OS implementation as well as the supplied test programs. In order to facilitate partial testing of the OS, you will also need the *supplied OS*, consisting of 8 `.vm` files (one for each OS class). Finally, you will need the supplied *VM Emulator*. This program will be used as the platform on which the actual test takes place. In order to start the project on the right foot, we also supply skeletal Jack files for each OS class.

Contract: Write the OS implementation and test it by running all the test programs and testing scenarios described below.

Recommended Testing Strategy

Here are the project materials:

- **Skeletal OS classes:** We supply one Jack file for each OS class. This file includes the "signatures" (interfaces) of all the functions and methods that should be implemented, with empty implementations. Your job is to provide the missing implementations according to the class API and the suggested algorithms.

- **Test programs:** For each OS class, we supply a separate test program written in Jack. In addition, we supply the Jack code of the *Pong* game as a "master" test.

We suggest that each class be developed and unit-tested in isolation. This can be done by compiling the OS class that you write and then putting the resulting `.vm` file in a directory that contains the other 7 `.vm` OS files and the `.vm` files of the respective test program. In particular, after implementing an OS class, you may follow these steps:

1) Copy your implemented OS class (Jack file) into the directory that contains the corresponding supplied test program (a collection of one or more Jack files);
2) Compile the entire directory using the *supplied* Jack Compiler;
3) Copy all the *supplied* OS `.vm` files (except the one that you have just compiled) into the directory.
4) At this point the directory should contain an executable program consisting of the eight `.vm` files related to the OS and one `.vm` file for each class in the test program;
5) Execute this program by opining the entire directory in the VM Emulator;
6) Proceed to test if the OS services are working properly according to the guidelines given below for each OS class.

After testing successfully each OS class in isolation, test your entire OS implementation using the *Pong* game. Put all your OS `.jack` files in the *Pong* directory, compile the directory, and execute the game in the VM Emulator. If the game works, that's pretty good.

Testing

Memory, Array, Math: In addition to the requisite `.jack` files, the test materials for each of these classes also include a `.tst` test script and a compare `.cmp` file for execution on the VM Emulator. To test your implementation of each one of these three OS classes, execute the given test scripts on the VM Emulator and make sure that the comparison ends successfully. Note that `Memory.alloc` and `Memory.deAlloc` are not fully tested, since a full test depends on internal implementation details not visible in user-level testing. Thus it is recommended to test these two methods using step-by-step debugging in the VM Emulator.

The remaining test programs do not include test scripts. They should be compiled and executed on the VM Emulator as is.

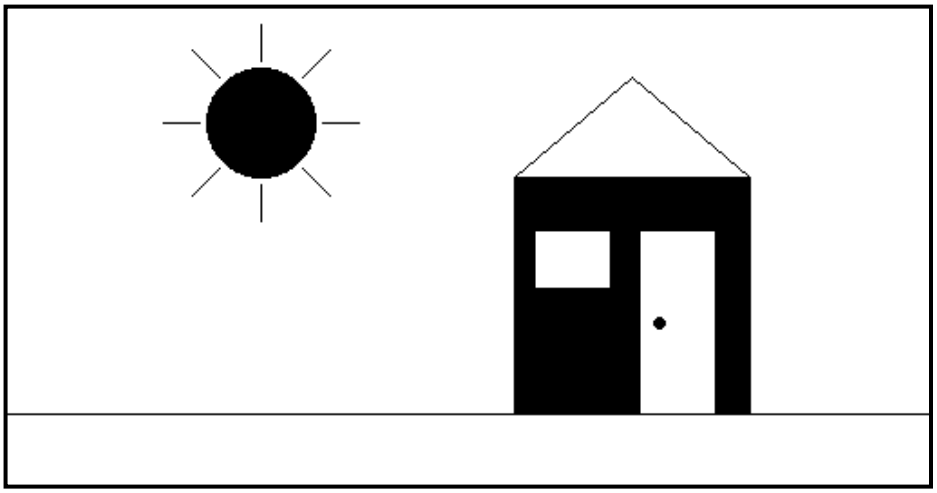
String: Execution of the corresponding test program should yield the following output:

```
new,appendChar: abcde
setInt: 12345
setInt: -32767
length: 12
charAt[2]: 99
setCharAt(2,'-'): ab-de
eraseLastChar: ab-d
intValue: 456
intValue: -32123
backSpace: 129
doubleQuote: 34
newLine: 128
```

Output: Execution of the corresponding test program should yield the following output:

```
A                                                                    B

0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
!#$%&'()*+,-./:;<=>?@[\]^_`{|}~
-12346789









C                                                                    D
```

Screen: Execution of the corresponding test program should yield the following output:



Keyboard: This OS class is tested using a test program that effects some program-user interaction. For each function in the `Keyboard` class (`keyPressed`, `readChar`, `readLine`, `readInt`) the program requests the user to press particular keys on the keyboard. If the function is implemented correctly and the correct keys are pressed, the program prints the text "`ok`" and proceeds to test the next function. If not, the program repeats the request for the same function. If all requests end successfully, the program prints '`Test ended successfully`', at which point the screen may look as follows:

```
keyPressed test:
Please press the 'Page Down' key
ok
readChar test:
(Verify that the pressed character is echoed to the screen)
Please press the number '3':
ok
readLine test:
(Verify echo and usage of 'backspace')
Please type 'JACK' and press enter:
ok
readInt test:
(Verify echo and usage of 'backspace')
Please type '-32123' and press enter: ");
ok

Test completed successfully
```

Sys: Only two functions in this class can be tested: `Sys.init` and `Sys.wait`. The supplied test program tests the `Sys.wait` function by requesting the user to press any key, waiting for two seconds (using `Sys.wait`) and then printing another message on the screen. The time that elapses from the moment the key is released until the next message is printed should be two seconds.

The `Sys.init` function is not tested explicitly. However, recall that it performs all the necessary OS initializations and then -- by definition -- calls the `Main.main` function of each test program. Therefore, we can assume that nothing would work properly unless `Sys.init` is implemented correctly. A simple way to test `Sys.init` in isolation is to run *Pong* using your `Sys.vm` file.