Eric Roberts
CS 208E

Handout #5
October 6, 2016

Algorithms

Algorithms

Eric Roberts
CS 208E
October 6, 2016

A Quick Introduction to JavaScript

- It is impossible to learn about programming without writing programs. In my book, I have decided—after trying several other possibilities—to use JavaScript as the programming language.
- One of the advantages of JavaScript is that it is by far the most common language for creating interactive content on the web. That means that it will certainly stick around.
- Another advantage is that JavaScript can be simplified into a language that is easy to learn and use without ever violating the rules of the language. You simply banish the messy parts of JavaScript and focus on the parts that are well designed.

Expressions in JavaScript

- As in most languages, computation in JavaScript is specified in the form of an ***expression***, which usually consists of ***terms*** joined together by ***operators***.
- Each term must be one of the following:
  – A constant (such as `3.14159265` or `"hello, world"`)
  – A variable name (such as `n1`, `n2`, or `total`)
  – A function call that returns a value (such as `sqrt`)
  – An expression enclosed in parentheses

Variables

- The simplest terms that appear in expressions are numeric constants and ***variables***. A variable is a placeholder for a value that can be updated as the program runs.
- A variable in JavaScript is most easily envisioned as a box capable of storing a value.

`answer`
$\boxed{42}$

- Each variable has the following attributes:
  – A ***name***, which enables you to tell different variables apart.
  – A ***value***, which represents the current contents of the variable.
- The name of a variable is fixed. The value changes whenever you ***assign*** a new value to the variable.

Variable Declarations

- It is good practice to ***declare*** a variable before you use it. The declaration sets the name of the variable and the initial value.
- The general form of a variable declaration is

```
var name = value;
```

where *name* is the name of the variable and *value* is an expression specifying the initial value.

- Most declarations appear as statements in the body of a function definition. Variables declared in this way are called ***local variables*** and are accessible only inside that function.
- Variables may also be declared outside of any function, in which case they are ***global variables***. The only global variables used in the book are constants written in upper case.

Operators and Operands

- Like most languages, JavaScript specifies computation using ***arithmetic expressions*** that closely resemble expressions in mathematics.
- The most common operators in JavaScript are the ones that specify arithmetic computation:

`+`	Addition	`*`	Multiplication
`-`	Subtraction	`/`	Division
		`%`	Remainder

- An operators usually appears between two subexpressions, which are called its ***operands***. Operators that take two operands are called ***binary operators***.
- The `-` operator can also appear as a ***unary operator***, as in the expression `-x`, which denotes the negative of `x`.

The Remainder Operator

- The only arithmetic operator that has no direct mathematical counterpart is `%`, which applies only to integer operands and computes the remainder when the first divided by the second:

```
14 % 5   returns   4
14 % 7   returns   0
7 % 14   returns   7
```

- The result of the `%` operator make intuitive sense only if both operands are positive. The examples in the book do not depend on knowing how `%` works with negative numbers.
- The remainder operator turns out to be useful in a surprising number of programming applications and turns up in several of the algorithms in Chapter 2.

Statement Types in JavaScript

- Statements in JavaScript fall into three basic types:
  - Simple statements
  - Compound statements
  - Control statements
- ***Simple statements*** are formed by adding a semicolon to the end of an expression, which is typically an assignment or a function call.
- ***Compound statements*** (also called ***blocks***) are sequences of statements enclosed in curly braces.
- ***Control statements*** fall into two categories:
  - ***Conditional statements*** that specify some kind of test
  - ***Iterative statements*** that specify repetition

Boolean Expressions

- JavaScript defines two types of operators that work with Boolean data: ***relational operators*** and ***logical operators***.
- There are six relational operators that compare values of other types and produce a `true`/`false` result:

`===`	Equals	`!==`	Not equals
`<`	Less than	`<=`	Less than or equal to
`>`	Greater than	`>=`	Greater than or equal to

For example, the expression `n <= 10` has the value `true` if `n` is less than or equal to 10 and the value `false` otherwise.

- There are also three logical operators:

`&&`	Logical AND	`p && q` means both `p` and `q`
`||`	Logical OR	`p || q` means either `p` or `q` (or both)
`!`	Logical NOT	`!p` means the opposite of `p`

Notes on the Boolean Operators

- Remember that JavaScript uses `=` for assignment. To test whether two values are equal, you must use the `===` operator.
- It is not legal in JavaScript to use more than one relational operator in a single comparison. To express the idea embodied in the mathematical expression

$$0 \leq x \leq 9$$

you need to make both comparisons explicit, as in

```
0 <= x && x <= 9
```

- The `||` operator means *either or both*, which is not always clear in the English interpretation of *or*.
- Be careful when you combine the `!` operator with `&&` and `||` because the interpretation often differs from informal English.

Short-Circuit Evaluation

- JavaScript evaluates the `&&` and `||` operators using a strategy called ***short-circuit mode*** in which it evaluates the right operand only if it needs to do so.
- For example, if `n` is 0, the right operand of `&&` in

```
n !== 0 && x % n === 0
```

is not evaluated at all because `n !== 0` is `false`. Because the expression

```
false && anything
```

is always `false`, the rest of the expression no longer matters.

- One of the advantages of short-circuit evaluation is that you can use `&&` and `||` to prevent execution errors. If `n` were 0 in the earlier example, evaluating `x % n` would cause a “division by zero” error.

The `if` Statement

The simplest of the control statements is the `if` statement, which occurs in two forms. You use the first form whenever you need to perform an operation only if a particular condition is true:

```
if (condition) {
   statements to be executed if the condition is true
}
```

You use the second form whenever you want to choose between two alternative paths, one for cases in which a condition is true and a second for cases in which that condition is false:

```
if (condition) {
   statements to be executed if the condition is true
} else {
   statements to be executed if the condition is false
}
```

The `while` Statement

The **`while`** statement is the simplest of JavaScript's iterative control statements and has the following form:

```
while ( condition ) {
    statements to be repeated
}
```

When JavaScript encounters a **`while`** statement, it begins by evaluating the condition in parentheses.

If the value of *condition* is **`true`**, JavaScript executes the statements in the body of the loop.

At the end of each cycle, JavaScript reevaluates *condition* to see whether its value has changed. If *condition* evaluates to **`false`**, JavaScript exits from the loop and continues with the statement following the closing brace at the end of the **`while`** body.

The `for` Statement

The **`for`** statement in JavaScript is a particularly powerful tool for specifying the control structure of a loop independently from the operations the loop body performs. The syntax looks like this:

```
for ( init ; test ; step ) {
    statements to be repeated
}
```

JavaScript evaluates a **`for`** statement as follows:

1. Evaluate *init*, which typically declares a ***control variable***.
2. Evaluate *test* and exit from the loop if the value is **`false`**.
3. Execute the statements in the body of the loop.
4. Evaluate *step,* which usually updates the control variable.
5. Return to step 2 to begin the next loop cycle.

Comparing `for` and `while`

The **`for`** statement

```
for ( init ; test ; step ) {
    statements to be repeated
}
```

is functionally equivalent to the following code using **`while`**:

```
init;
while ( test ) {
    statements to be repeated
    step;
}
```

The advantage of the **`for`** statement is that everything you need to know to understand how many times the loop will run is explicitly included in the header line.

Exercise: Reading `for` Statements

Describe the effect of each of the following **`for`** statements:

1. `for (var i = 1; i <= 10; i++)`

2. `for (var i = 0; i < N; i++)`

3. `for (var n = 99; n >= 1; n = n - 2)`

4. `for (var x = 1; x <= 1024; x = x * 2)`

Writing Functions

- The general form of a function definition is

```
function name (parameter list) {
    statements in the function body
}
```

  where *name* is the name of the function, and *parameter list* is a list of variables used to hold the values of each argument.

- You can return a value from a function by including a **`return`** statement, which is usually written as

```
return expression;
```

  where *expression* is an expression that specifies the value you want to return.

Functions Involving Control Statements

- The body of a function can contain statements of any type, including control statements. As an example, the following function uses an **`if`** statement to find the larger of two values:

```
function max(x, y) {
   if (x > y) {
      return x;
   } else {
      return y;
   }
}
```

- As this example makes clear, **`return`** statements can be used at any point in the function and may appear more than once.

The `factorial` Function

- The ***factorial*** of a number *n* (which is usually written as *n*! in mathematics) is defined to be the product of the integers from 1 up to *n*. Thus, 5! is equal to 120, which is 1×2×3×4×5.
- The following function definition uses a **`for`** loop to compute the factorial function:

```
function fact(n) {
   var result = 1;
   for (var i = 1; i <= n; i++) {
      result = result * i;
   }
   return result;
}
```

Functions and Algorithms

- Functions are critical to programming because they provide a structure in which to express algorithms.
- Algorithms for solving a particular problem can vary widely in their efficiency. It makes sense to think carefully when you are choosing an algorithm because making a bad choice can be extremely costly.
- The next few slides illustrate this principle by implementing two algorithms for computing the ***greatest common divisor*** of the integers *x* and *y,* which is defined to be the largest integer that divides evenly into both.

The Brute-Force Approach

- One strategy for computing the greatest common divisor is to count backwards from the smaller value until you find one that divides evenly into both. The code looks like this:

```
function gcd(x, y) {
   var guess = x;
   while (x % guess != 0 || y % guess != 0) {
      guess--;
   }
   return guess;
}
```

- This algorithm must terminate for positive values of **`x`** and **`y`** because the value of **`guess`** will eventually reach 1. At that point, **`guess`** must be the greatest common divisor.
- Trying every possibility is called a ***brute-force strategy***.

Euclid's Algorithm

- If you use the brute-force approach to compute the greatest common divisor of 1000005 and 1000000, the program will take a million steps to tell you the answer is 5.
- You can get the answer much more quickly if you use a better algorithm. The Greek mathematician Euclid of Alexandria described a more efficient algorithm 23 centuries ago, which looks like this:

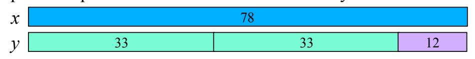
```
function gcd(x, y) {
   while (x % y != 0) {
      var r = x % y;
      x = y;
      y = r;
   }
   return y;
}
```

How Euclid's Algorithm Works

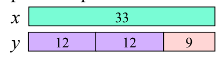
- If you use Euclid's algorithm on 1000005 and 1000000, you get the correct answer in just two steps, which is much better than the million steps required by brute force.
- Euclid's great insight was that the greatest common divisor of *x* and *y* must also be the greatest common divisor of *y* and the remainder of *x* divided by *y*. He was, moreover, able to prove this proposition in Book VII of his *Elements*.
- It is easy to see how Euclid's algorithm works if you think about the problem geometrically, as Euclid did. The next slide works through the steps in the calculation when *x* is 78 and *y* is 33.

An Illustration of Euclid's Algorithm

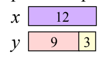
Step 1: Compute the remainder of 78 divided by 33:



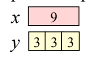
Step 2: Compute the remainder of 33 divided by 12:



Step 3: Compute the remainder of 12 divided by 9:



Step 4: Compute the remainder of 9 divided by 3:



Because there is no remainder, the answer is 3: