

Calculators and Tools for Principles of Computer Networks

Joshua Crotts¹

Email: ¹ljcrotts@uncg.edu

November 24, 2020

Abstract

As someone with strong interests in computer science pedagogy, there are some topics presented throughout classes that, while informative and fulfilling in theory, can be difficult to visualize without an aid. Learning is an important piece of computer science, complemented by both the traditional classroom setting and application. When doing preliminary research for this program/project, I discovered that there is a lack of quality software or user-friendly websites that provided the calculators that I needed. As such, I built my own versions, intended for the Principles of Computer Networks (CSC-477/677) students. This report describes the tools and small calculators that I have developed during this semester. Originally, these were separate programs, but for the purposes of future redistribution, they are bundled in a GitHub repository with instructions on how to use it.

1 Introduction

Principles of Computer Networks for undergraduates and graduates at the University of North Carolina Greensboro introduces students to several key concepts about networking fundamentals, with topics ranging from high-level descriptions of the Open Systems Interconnection (OSI) model down to essential algorithms that play crucial roles in data transformation and transmission. Furthermore, many calculations involving IP addresses exist, allowing students to practice how data is encapsulated, transmitted, and changed over a network, and moreover how it progresses down the OSI stack. Some of these calculations are quite challenging or cumbersome to perform by hand, one prime example being the binary division, which is performed at the network layer for cyclic redundancy checks (CRC). Long division is a topic many are taught in elementary school, but lose the grasp of it through the transition to higher education, not to mention how needlessly inconvenient it is to do by hand. Another example that

pertains heavily to the network simulator project is the transport and network layer checksum. Performing such a calculation on a large string of hex characters is prone to mistakes with needing to convert from hexadecimal to decimal, or to another numbering system. This is why computer programs and automation exist in the first place. Therefore, when students are creating their checksum portion of the simulator, it is convenient to have a piece of code that allows for easy verification that they are on the right track. So, I took it upon myself to build such a tool. Other computational resources were created with the same idea in mind: to reduce the time it takes to do the computation by hand. However, while automation makes completing the task easy, it abstracts the core details of the algorithms used to perform the calculation, leaving some clueless as to *how* something works, as they only know that it does work. This document lists the structural, functional, and logical characteristics of my program, along with rebuilding it on multiple environments.

```
Welcome to a small tool for CSC-677.
1. Binary Division Calculator (Long Division Bits)
2. Subnet Block Calculator
3. Classful Subnet Generator
4. Checksum Calculator (For Hex Strings)
5. Random Value Generation
6. Hamming Distance
7. Parity Bit Checker
8. IP Subnet Calculator
9. Generate IP Information
10. EXIT
Choose an option: █
```

Figure 1: Initial window displayed upon opening the program.

2 Users Manual

Our tool contains nine sub-programs, listed in no particular order (however enumerated as-is in the project):

1. Binary Division Calculator
2. Subnet Block Calculator
3. Classful Subnet Generator

4. Checksum Calculator
5. Random Value Generation
6. Hamming Distance
7. Parity Bit Checker
8. IP Subnet Calculator
9. Generate IP Information

For Windows users, a runnable executable (.EXE) file is included in the repository. Other users, however, will require the tools laid out in the following section about recompiling, due to C's lack of portability.

3 Recompiling

To recompile the program, a C compiler is required. This program was developed mostly on a Windows machine with the `gcc` compiler, but was tested on MacOS with `clang`, as well as Linux Mint with `gcc` once again. A `makefile` is also included to make compiling the project easier.

1. Download a compatible C compiler. As suggested, `gcc` or `clang` work well for this. A good C compiler for Windows systems is MinGW. Linux users generally have a C compiler pre-installed.
2. Clone the repository <https://github.com/JoshuaCrotts/CSC-677-Tools>. Ideally, this should be placed somewhere easily accessible by a development environment.
3. Navigate to the directory in a terminal or command line, then type `make` or `makefile`. The project should compile successfully given that no alterations are made beforehand.
4. Run the project by executing the command `./Network.exe`, or `./Network`. The former only works on Windows systems.

4 Algorithms and Code Details

Some may wonder why I chose C instead of something simpler like Java when developing this project. There are two primary reasons: first, a lot of networking (and low-level) code is written in C/C++. It is a good idea to introduce this less-abstract code to students in a class of this nature. Second, most of the operations and algorithms manipulate directly on bits and numbers. Java makes it simple to convert between notations (say, for instance, using a String to hold a binary string), but at the same time is slower as a result. The programs are

not slow enough to where they are noticeable by the average user, but working with raw numbers helps better grasp why the algorithms work at that level. Besides, the actual algorithms that many websites (and especially those used in real networking algorithms and protocols) do not work on Strings; everything is encoded via numbers for speed purposes. In this section, we will describe the nine sub-programs developed, along with the predominant algorithm, and a few examples and screenshots with the working program to provide users an idea of how it should be used. For reference, the operators \gg and \ll refer to a bitwise shift right and left respectively. Similar to the C equivalents, these shift the bits of a binary number either to the right or left by one, respectively. Moreover, $\&$ and $|$ refer to bitwise AND and inclusive-OR operators. Additionally, instead of the tilde (\sim) operator used for negation, we will use the more familiar logic negation (\neg) operator for bitwise negation. Lastly, unlike its C counterpart, the exclusive-OR will instead be \oplus .

4.1 Binary Division Calculator (Long Division Bits)

Performing long division on binary numbers is difficult and tedious for humans, but is extensively required for cyclic redundancy checks as aforementioned. Below is the algorithm used:

Algorithm 1 Performs Binary Division on Two Binary Numbers.

Ensure: x is the dividend, y is divisor

```

1: procedure BINARYDIVISION( $x, y$ )
2:    $x_d \leftarrow \lfloor \log_2(x) + 1 \rfloor$  ▷ Binary digits in  $x$ 
3:    $y_d \leftarrow \lfloor \log_2(y) + 1 \rfloor$  ▷ Binary digits in  $y$ 
4:    $r \leftarrow x \gg (x_d - y_d)$  ▷ Current dividend being used.
5:    $q \leftarrow 1$  ▷ The quotient starts at 1.
6:
7:   for  $i \leftarrow (y_d - 1)$  to  $(x_d - 1)$  do
8:      $res \leftarrow r \oplus y$  ▷ XOR is subtraction on binary numbers.
9:      $res_d \leftarrow \lfloor \log_2(res) + 1 \rfloor$ 
10:    while  $r_d \neq y_d$  do ▷ Ensure dividend size fits in divisor
11:       $i \leftarrow i + 1$ 
12:      if  $i \geq x_d$  then
13:        break
14:       $next_d \leftarrow (x \gg (x_d - i - 1)) \& 1$ 
15:       $res \leftarrow res \ll 1$  ▷ Move results over by 1 binary digit.
16:       $q \leftarrow q \ll 1$ 
17:       $res \leftarrow res \mid next_d$  ▷ Store next digit in result.
18:       $res_d \leftarrow res_d + 1$ 
19:
20:      if  $res_d \neq y_d$  then
21:         $q \leftarrow q \mid 0$ 
22:      else
23:         $q \leftarrow q \mid 1$ 
24:       $r \leftarrow res$ 
25:

```

Admittedly, this algorithm has a lot of binary arithmetic and bitwise operators that are somewhat esoteric for non-C programmers. The idea behind it, though, thankfully is not: ensure that the dividend x is greater than or equal to y . Subtract the divisor from the dividend. Store a 1 in the quotient if the divisor fits into the dividend, and 0 otherwise. Then, extend the dividend by the next binary digit in line to its right. It resembles regular long-division. One added bonus to the program, though, is that the program prints the result in polynomial notation (ex. if our quotient is 10011, we return $x^4 + x + 1$). For simplicity's sake, we will not include the algorithms for converting the result into a polynomial string.

```
Enter the dividend in decimal: 288
Enter the divisor in decimal: 13
Quotient: 111101 => x^5 + x^4 + x^3 + x^2 + 1
Remainder: 1 => 1
Continue? 1 for yes, 0 for no: █
```

Figure 2: Binary division with decimal numbers 288 (100100000) and 13 (1101).

4.2 Block Calculator

Next, we include a personal favorite: block calculations. Given a starting address, a subnet, and a block size (along with the number of addresses in each block), we can compute the address ranges for each block. When entering them into the program, the number of addresses in each block must be entered in reverse sorted order, meaning the largest block is entered first, with the smallest last. This ensures that the largest block has its addresses populated before working its way down to the smaller ones. Since this program requires a lot of output to the screen, we will use the function `PRINT` to simulate printing to the output window/screen.

Algorithm 2 Computes Address Ranges for Subnet Blocks.

Ensure: ip is an IP address in hex, s is a subnet in CIDR notation, b_c is the number of blocks used, B is a list of values in reverse sorted order denoting how many addresses are in a block.

```
1: procedure BLOCK( $ip, s, b_c, B$ )
2:    $min_a \leftarrow ip$ 
3:    $max_a \leftarrow ip$ 
4:   for  $i \leftarrow 0$  to  $b_c - 1$  do
5:      $b \leftarrow B[i]$ 
6:      $b_s \leftarrow pow(2, \lceil \log_2 b \rceil)$ 
7:      $mask \leftarrow 32 - \log_2 b_s$ 
8:      $max_a \leftarrow max_a + b_s - 1$ 
9:     PRINT( $min_a$ )
10:    PRINT( $max_a$ )
11:     $max_a \leftarrow max_a + 1$ 
12:     $min_a \leftarrow max_a$ 
13:     $i \leftarrow i + 1$ 
```

Some details are omitted for clarity's sake, such as those that require input from the user, and will continue to be omitted from future algorithms. This algorithm uses our list of block sizes B to compute how many addresses are given to the i^{th} block. Address sizes may only be in increments of powers of 2, so a small offset is performed to round the size up to the next nearest power of 2. From there, we simply add this value to the current minimum address to get the maximum address for this range, and print the information to the user. At the end of the for loop, the addresses are reset to account for the next block in B .

```

Enter your beginning address in hex WITH the leading 0x prefix (e.g. 0xFFABCD45)
: 0xffabcd45

Enter your beginning subnet in decimal (e.g. 24, 25, ...): 25

Enter the number of blocks to use in decimal: 3

When entering the subblock addresses in decimal, enter them in reverse order starting with the largest block.

Total addresses: 120.
Enter the subblock 1 address count: 120
Enter the subblock 2 address count: 60
Enter the subblock 3 address count: 10
Minimum address: 255.171.205.69
Maximum address: 255.171.205.196
The starting address for block 1 is 255.171.205.69 /25
The ending address for block 1 is 255.171.205.196 /25
The starting address for block 2 is 255.171.205.197 /26
The ending address for block 2 is 255.171.206.4 /26
The starting address for block 3 is 255.171.206.5 /28
The ending address for block 3 is 255.171.206.20 /28

```

Figure 3: IPv4 address of 255.171.205.69, subnet 25, three blocks of size 120, 60, and 10 addresses respectively.

4.3 Classful Subnet

Our next algorithm is rather simple but helpful in certain situations. Given a classful hexadecimal address c , and a subnet mask m , we can compute the number of usable sub-networks as follows:

Algorithm 3 Computes Number of Usable Sub-Networks in Classful Subnet and Subnet Mask.

Ensure: c is a hexadecimal number matching a classful A , B , or C subnet, m is a hexadecimal number matching a valid subnet mask.

- 1: **procedure** CLASSFUL(c, m)
 - 2: $r = c \oplus m$
 - 3: $r = r / (-r \& r);$
-

4.4 Checksum

Our third program is likely the hardest program to understand at first, but by far the most useful for Principles of Computer Networks students. Given a hex string l , where $|l|$ is a multiple of four, we can compute the transport/network layer checksum as follows:

Algorithm 4 Computes the checksum of a hexadecimal string.

Ensure: l is a hex string of characters where $|l|$ is a multiple of four.

```

1: procedure CHECKSUM( $l$ )
2:    $c \leftarrow 0$  ▷ Checksum accumulator.
3:    $s_l \leftarrow |l| - 1$ 
4:   for  $i \leftarrow 0$  to  $s_l - 1$  do
5:      $substr \leftarrow \text{SUBSTRING}(l, i, i + 4)$ 
6:      $dec \leftarrow \text{HEXTODEC}(substr)$  ▷ Convert  $substr$  to decimal.
7:      $c \leftarrow c + dec$ 
8:      $r \leftarrow c \gg 16$ 
9:      $c \leftarrow c \& 0xffff$ 
10:     $c \leftarrow c + r$ 
11:     $c \leftarrow 0xffff - c$ 

```

At the start of the paper, it was stated that the idea of programming in C was to eliminate performance overhead by using strings, but this program goes against the statement by using them. However, the string is only used for input; the SUBSTRING procedure is hand-written and has little overhead. All other computations are handled via bit and numeric operations. Now, let us describe the program: we extract two bytes at a time (16 bits), and calculate the decimal equivalent of those two bytes. This is then added to a variable c which represents our running checksum total. Once all bytes are accounted for, we grab any bits beyond the sixteenth, those of which are stored in a variable r denoting the remainder. Then, our checksum is truncated to sixteen bits using a bitwise AND. At this point, our remainder is re-added to our checksum. Finally, we subtract our checksum from the maximum value of an unsigned 16 bit number ($0xffff$, or 65535), which becomes our checksum. This process ensures that our result never exceeds sixteen bits in width.

```
Enter your hex string WITHOUT the hex prefix with length multiple of 4 (AB12CCDD
): ABCD12340000FFFF
Remainder: 1
Truncating checksum to 16 bits is 48640
The checksum in decimal is 16894.
The checksum in hex is 0x41fe.

Continue? 1 for yes, 0 for no: █
```

Figure 4: Checksum computation.

4.5 Random Data Generation

Sometimes, it is helpful to have random data on the fly to test programs with. This sub-program is, naturally, a sub-program within a sub-program, meaning there are multiple features. The first being a valid IPv4 address generator. The .c file describes IP addresses that are invalid at the top, and are matched against this when generating a value. In our pseudocode, this list is described as *B*. The second algorithm generates a random MAC address, valid or not. Our last feature, which is not documented in an algorithm, allows the user to enter an IP address in standard form as a string (xxx.xxx.xxx.xxx). From there, it is outputted in hexadecimal form, thereby allowing for easy access to multiple pieces of the program that require the input of an IP address to be a hex value instead of a string (which are cumbersome to parse).

4.6 Hamming Distance

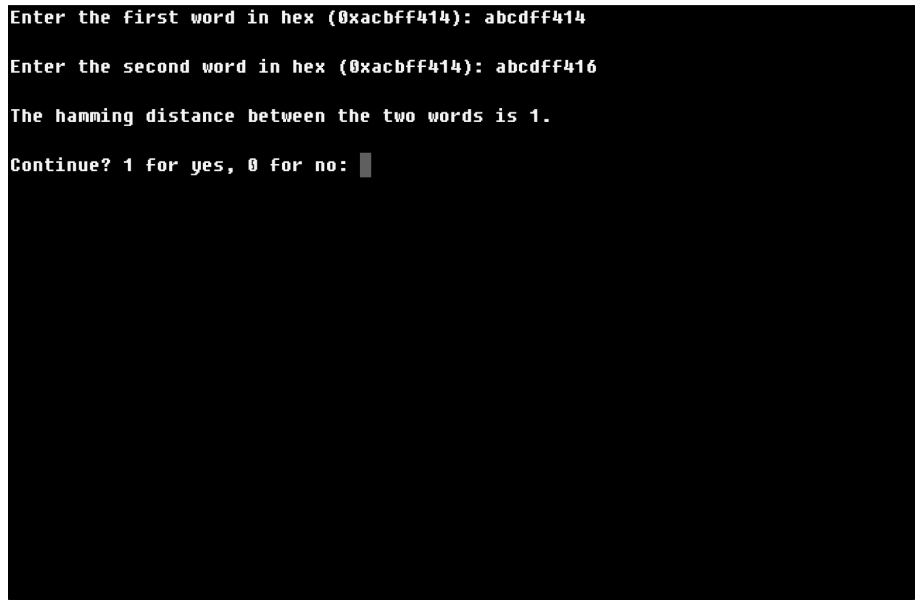
When receiving data over a network, it can be helpful to compare data to see if there are any alterations that occurred during transit. The Hamming Distance algorithm is a simple but efficient method for determining how many bits are incorrect (i.e. flipped):

Algorithm 5 Computes Hamming Distance of Two Numbers.

Ensure: a and b are valid numbers.

- 1: **procedure** HAMMING(a, b)
 - 2: $d \leftarrow \text{COUNTSETBITS}(a \oplus b)$
-

All we need to do is perform a bitwise exclusive-OR on the two numbers a and b . This operation toggles all bits that differ between the two values. Consequently, each set bit represents where a disparity occurs. Hence, $a = b$ if and only if $a \oplus b = 0$, because bits that do not match are flipped via the XOR operator. \square



```
Enter the first word in hex (0xacbfff414): abcdff414
Enter the second word in hex (0xacbfff414): abcdff416
The hamming distance between the two words is 1.
Continue? 1 for yes, 0 for no: █
```

Figure 5: Hamming distance checker with two words that are off by one bit only.

4.7 Parity Bit Checker

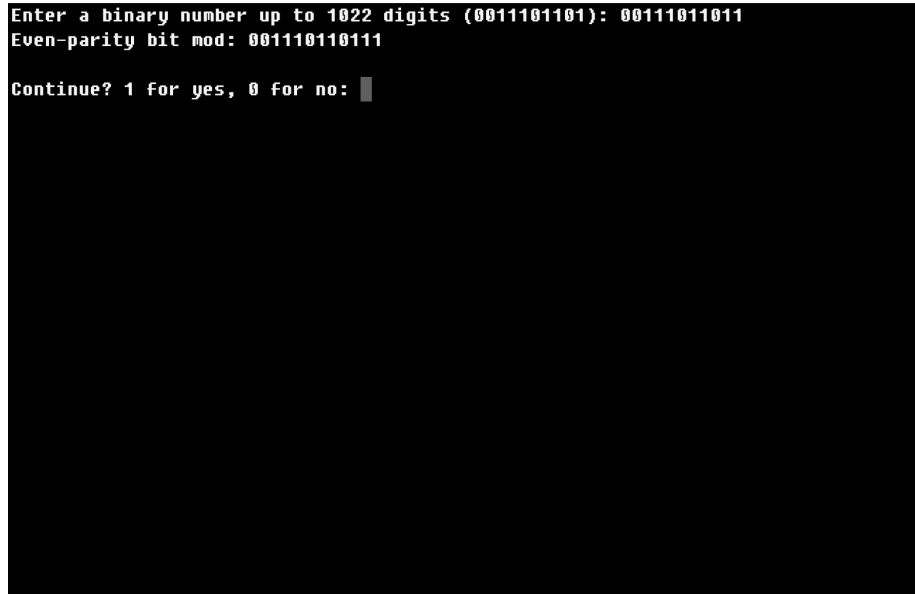
Parity bits are, like hamming codes, used in error detection. However, a simpler approach is taken: we append a 0 to the result if there is “even parity” in the number. Otherwise, we append 1.

Algorithm 6 Computes the Parity Bit Modification for a Number n .

Ensure: n is a valid number.

```
1: procedure PARITY( $n$ )
2:    $s \leftarrow \text{COUNTSETBITS}(n)$ 
3:   if  $s$  is even then
4:      $s \leftarrow (s \ll 1) \mid 0$ 
5:   else
6:      $s \leftarrow (s \ll 1) \mid 1$ 
```

First, as aforementioned, the number of set bits are calculated. If this number is even, then we do not need to adjust the number for “even parity”, because the number already *has* even parity (this does not suggest that the number remains unaltered, though). However, if the number of set bits is odd, then we append a 1 to the far-right (least-significant bit) of the number. Note that we do not *flip* the LSB; rather we shift the number, and append the appropriate result.



```
Enter a binary number up to 1022 digits (0011101101): 0011101101
Even-parity bit mod: 001110110111
Continue? 1 for yes, 0 for no: █
```

Figure 6: Parity bit checker - adds one bit to the result.

4.8 IP Subnet Calculator

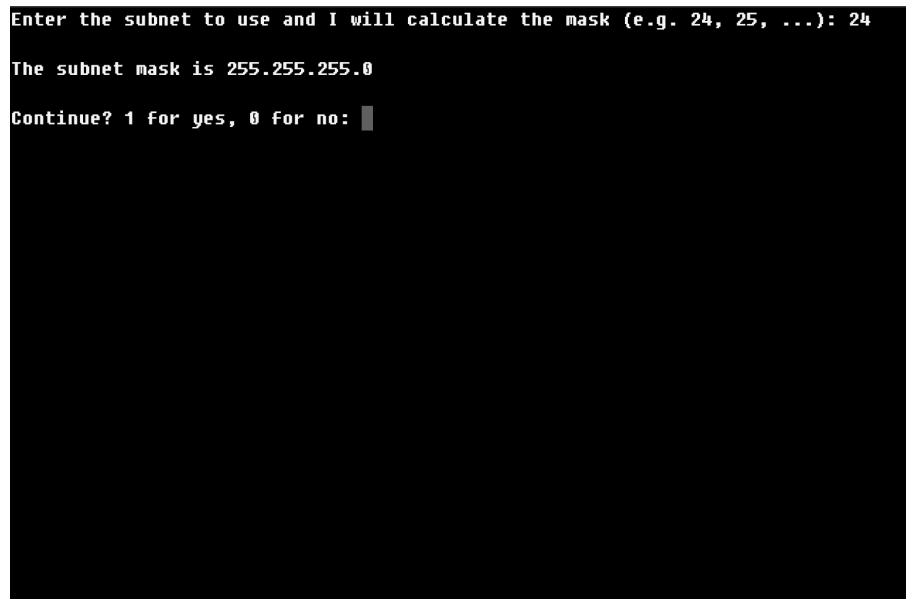
While a simple program, knowing what the actual subnet mask is for a given subnet in CIDR notation is helpful if one does not remember how to compute it on the fly.

Algorithm 7 Computes Subnet Mask when Given Subnet in CIDR Notation.

Ensure: c is a valid CIDR subnet (between 1 and 32)

- 1: **procedure** SUBNETMASK(c)
 - 2: $s \leftarrow 0xffffffff \& \neg(pow(2, 32 - c) - 1)$
-

The procedure is rather straightforward: the maximum value for a 32-bit integer (an IPv4 address or subnet mask) is $0xffffffff$ in hex. We bitwise AND this by the negation of our subnet when raised to the power of $32 - c$, where c is our CIDR subnet. For instance, suppose $c = 24$. We know $2^{32-24} = 2^8 = 256$. $256 - 1 = 255 = 0xff$. A bitwise negation of $0xff$ in a 32-bit integer is $0xffffffff00$, which in standard IPv4 addressing is 255.255.255.0. We know that when $c = 24$, this is a classful C subnet, which matches our calculation. The bitwise AND is to ensure that our number stays within the bounds of a 32-bit integer, unsigned or signed. \square



```
Enter the subnet to use and I will calculate the mask (e.g. 24, 25, ...): 24
The subnet mask is 255.255.255.0
Continue? 1 for yes, 0 for no: █
```

Figure 7: Computes the mask of a CIDR /24 - class C mask.

4.9 Generate IP Data

Our last section discusses a mini-program that, in and of itself, is much larger than the others discussed. There are several components to an IPv4 address that, when provided a subnet, we may want to compute. These include:

1. Subnet mask
2. Binary mask

3. Wildcard mask (negation of subnet mask)
4. CIDR notation of subnet
5. Network address
6. Number of hosts
7. Number of usable hosts
8. First host address
9. Last host address
10. Broadcast address
11. IP Classification (A, B, C, D)

Now, because most of these procedures are sub-procedures themselves, we will not go into the details of any algorithms, as most are one-line bitwise operations that do not add much contribution to the paper. Though, we will still provide an example of its usage:

```
Enter your IP in hex WITH the leading 0x prefix: 0xc0a80101
Enter your subnet in CIDR notation (ex. /23): 24

IPv4 Address: 192.168.1.1
Subnet: 255.255.255.0
Binary Netmask: 11111111.11111111.11111111.00000000.
Wildcard Mask: 0.0.0.255
CIDR Notation: /24
Network Address: 192.168.1.0
Number of Hosts: 256
Number of Usable Hosts: 254
First Host Address: 192.168.1.1
Ending Host Address: 192.168.1.254
Broadcast Address: 192.168.1.255
IP Classification (Classful): C (192.0.0.0 - 223.255.255.255)

Continue? 1 for yes, 0 for no: █
```

Figure 8: Computes the IP data for 192.168.1.1.

5 Conclusion

In summary, having a visual aid when trying to perform computations by hand is a very helpful advantage. Sometimes, and from what was discovered during

research for this project, those found online do not fit best with what students come across in CSC-477/677. Furthermore, websites require the internet for access, whereas this program only requires a command line interface (provided that the program has been previously downloaded and compiled). In addition, many websites do not provide the internal documentation or algorithms that run the code that students see, so while they may have a solution to whatever problem they are working on, they are unaware of the specifics.