What is Computer Science?

An Information Security Perspective

Daniel Page ⟨dan@phoo.org⟩ and Nigel P. Smart ⟨csnps@bristol.ac.uk⟩

git # 148def3 @ 2018-07-11

HOW LONG IS A PIECE OF STRING?

Recall that in Chapter 4 we described how computers only understand a fixed set of instructions that can operate only on fixed types of operand; at the lowest level, instructions typically only operate on operands which are numbers. So, for example, most computers understand how to add together numbers: in this case we would say that a number is a **native** type of data. However, it is not common for a computer to understand types of data more complicated than this. For example a computer does not have a native understanding of emails; there is no instruction that tells it to "search for the text X in the email Y". In order to write a program that deals with non-native types of data like this, we need to decide on a representation for it using a **data structure**.

The design of data structures and the algorithms that perform operations on them is a fundamental subject that underpins almost every aspect of Computer Science. Essentially, a data structure allows us to look at numbers and interpret them as something else. Sometimes the data structure is simply a description of how numbers should be arranged so we can interpret them correctly; sometimes we add extra information or **meta-data** which allows us to inspect and alter the data structure. Interested in computer graphics? You need data structures (e.g., representations of images, or points in 3D space) and algorithms to operate on them (e.g., change the colour of an image, or rotate a 3D scene). Interested in computer networks? You need data structures (e.g., representations of connectivity between computers on the network) and algorithms to operate on them (e.g., find a path from computer X to computer Y). Interested in computer security? You need data structures (e.g., representations of very large numbers) and algorithms to operate on them (e.g., add together X and Y modulo Z). In all cases, efficiency is key: if we use a more efficient data structure, we might reduce the time taken to perform a given operation, or the amount of memory required to store the data.

Our focus here is on **strings**. The term string crops up in a lot of places, but generically means a sequence of things. For example, we string together a sequence of words to make a sentence, protein is made from a string of many amino acids and so on. In Computer Science, a string is a sequence of **characters** and one of the most fundamental data structures after the native types of data which a computer can operate on. If you think about it, an email is just a string of characters, the files created by a word processor could be thought of as similar strings, the messages a computer displays so it can communicate with the user are also strings, and so on. The aim is to show that even with something that seems so simple, there are *many* options for useful data structures and that our choice has a *major* impact on the algorithms used to manipulate them.

1 String data structures

As we saw in Chapter 4, the memory of a computer is where it stores data in the long term. The computer loads the data into an accumulator to operate on it, and stores the result back in memory afterwards. We modelled memory using a sequence called MEM and stored decimal numbers in each element. Since MEM is large, it is convenient to avoid writing out the whole sequence. To allow this, we will write the addresses of elements above the elements themselves. For example, imagine we have

$$\begin{array}{rcccccccccc} i & = & \ldots, & 3, & 4, & 5, & 6, & 7, & \ldots & & \\ \mathsf{MEM} & = & \langle\ \ldots, & 104, & 101, & 108, & 108, & 111, & \ldots & \rangle \end{array}$$

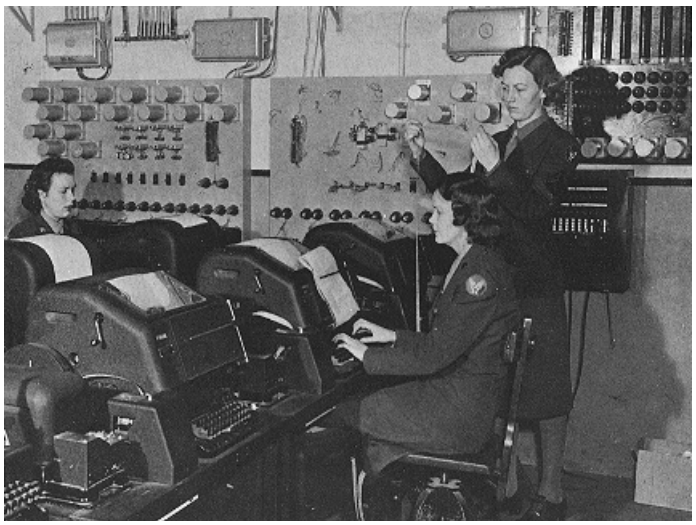

Figure 1: *A teletype machine being used by UK-based Royal Air Force (RAF) operators during WW2 (public domain image, source: http://en.wikipedia.org/wiki/File:WACsOperateTeletype.jpg).*

If MEM has n elements in total, we have missed out the elements at addresses $0 \ldots 2$ and those at addresses $8 \ldots n-1$. We are only interested in addresses #3, #4, #5, #6 and #7 which have the values 104, 101, 108, 108 and 111. Our goal is to represent strings within MEM but to do this, we need to solve two problems: first how to represent characters, and second how to represent strings.

1.1 Problem #1: representing characters

The first problem is that MEM can only store numbers, and the computer it exists within can only really process numbers. To allow it to deal with characters, we need some way to represent them numerically. Basically we just need to translate from one to the other. More specifically, we need two functions: $\textsc{Ord}(x)$ which takes a character x and gives us back the corresponding numerical representation, and $\textsc{Chr}(y)$ which takes a numerical representation y and gives back the corresponding character. But how should the functions work? Fortunately, people have thought about this problem for us and provided standards we can use. One of the oldest is the **American Standard Code for Information Interchange (ASCII)** [1], pronounced "ass key".

ASCII has a rich history, but was developed to permit communication between early teleprinter devices. These were like a combination of a typewriter and a telephone, and were able to communicate text to each other before innovations such as the fax machine. Later, but long before monitors and graphics cards existed, similar devices allowed users to send input to early computers and receive output from them. Figure 2 shows the 128-entry ASCII table which tells us how characters are represented as numbers. Of the entries, 95 are printable characters we can instantly recognise (including *SPC* which is short for "space"). There are also 33 others which represent non-printable control characters: originally, these would have been used to control the teleprinter rather than to have it print something. For example, the *CR* and *LF* characters (short for "carriage return" and "line feed") would combine to move the print head onto the next line; we still use these characters to mark the end of lines in text files. Other control characters also play a role in modern computers. For example, the *BEL* (short for "bell") characters play a "ding" sound when printed to most UNIX terminals, we have keyboards with keys that relate to *DEL* and *ESC* (short for "delete" and "escape") and so on.

You can test this out by issuing the command

```
echo -e '\x07'
```

in a BASH terminal. It asks `echo` to display the ASCII character $07_{(16)} = 7_{(10)} = \textit{BEL}$ on the terminal, which *should* mean a sound is produced (or perhaps a visual indicator if the sound is disabled). The `-e` option is quite important, because it means that `\x07` is interpreted as an ASCII code rather than a normal string. Try this with some other examples. For instance, what would you expect

```
echo -e 'hello\x08world'
```

to produce?

Since there are 128 entries in the table, ASCII characters can be and are represented by bytes as 8-bit numbers. However, notice that $2^7 = 128$ and $2^8 = 256$ so in fact we *could* represent 256 characters: essentially one of the

y Ord(x)	Chr(y) x	y Ord(x)	Chr(y) x	y Ord(x)	Chr(y) x	y Ord(x)	Chr(y) x
0	*NUL*	1	*SOH*	2	*STX*	3	*ETX*
4	*EOT*	5	*ENQ*	6	*ACK*	7	*BEL*
8	*BS*	9	*HT*	10	*LF*	11	*VT*
12	*FF*	13	*CR*	14	*SO*	15	*SI*
16	*DLE*	17	*DC*1	18	*DC*2	19	*DC*3
20	*DC*4	21	*NAK*	22	*SYN*	23	*ETB*
24	*CAN*	25	*EM*	26	*SUB*	27	*ESC*
28	*FS*	29	*GS*	30	*RS*	31	*US*
32	*SPC*	33	!	34	"	35	#
36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+
44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;
60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C
68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K
76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S
84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[
92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c
100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k
108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s
116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{
124	|	125	}	126	~	127	*DEL*

Figure 2: *A table describing the printable ASCII character set.*

bits is not used by the ASCII encoding. Specific computer systems sometimes use the unused bit to permit use of an "extended" ASCII table with 256 entries; the extra characters in this table can be used for some special purpose. For example, foreign language characters are often defined in this range (e.g., é or ø). However, the original use of the unused bit was as an error detection mechanism: more specifically, it represents the parity bit from the even parity code in Chapter 2.

Another use for extended ASCII characters is **ASCII art** [2], popular in an era when plain text was the *only* viable display medium. Examples archived at

http://www.textfiles.com/

were often used by **Bulletin Board Systems (BBSs)** [4], which used the telephone system and MODEMs to form an early form of computer network: text was used almost exclusively to compensate for the relatively slow speed of data transfer. An excellent documentary at

http://www.bbsdocumentary.com/

chronicles the technology and personalities involved.

There are now various ASCII art editors online, such as

http://www.asciigraffiti.com/

Forget GIMP, which we used in Chapter 2, or other modern image manipulation software: take a trip into the past, and see how hard it is to draw images using text!

Given the table, we can see, for example, that Chr(104) = 'h', i.e., if we see the number 104 then this represents the character 'h'. Conversely we have that Ord('h') = 104. Although in a sense any consistent translation between characters and numbers like this would do, ASCII has some useful properties. Look specifically at the alphabetic characters:

- Imagine we want to test if one character x is alphabetically before some other y. The way the ASCII translation is specified, we can simply compare their numeric representation. If we find $\text{Ord}(x) < \text{Ord}(y)$ then the character x is before the character y in the alphabet. For example 'a' is before 'c' because

$$\text{Ord}(\text{'a'}) = 97 < 99 = \text{Ord}(\text{'c'}).$$

- Imagine we want to convert a character x from lower-case into upper-case. The lower-case characters are represented numerically as the contiguous range $97 \ldots 122$; the upper-case characters as the contiguous range $65 \ldots 90$. So we can covert from lower-case into upper-case simply by subtracting 32. For example

$$\text{Chr}(\text{Ord}(\text{'a'}) - 32) = \text{'A'}.$$

Armed with the ASCII table and the new translation method, we can read new meaning into the contents of MEM. As well as writing the addresses of elements above the elements, we can write the ASCII translation of each element (i.e., the character that each numeric element represents) below this, on an extra line. For example:

i	=	$\ldots$,	3,	4,	5,	6,	7,	$\ldots$	
MEM	=	⟨ $\ldots$,	104,	101,	108,	108,	111,	$\ldots$	⟩
Chr(MEM)	=	$\ldots$,	'h',	'e',	'l',	'l',	'o',	$\ldots$	

We can now see that addresses $3 \ldots 7$ hold the string "hello" if we interpret the memory content as ASCII characters. Just to show that this really *is* how things work, consider this short experiment using BASH:

```
bash$ cat > A.txt
hello
bash$ cat A.txt
hello
bash$ cat A.txt | od -Ad -tu1 | cut -c 9-
104 101 108 108 111  10

bash$
```

First we create a file called `A.txt` by typing the characters 'h', 'e', 'l', 'l' and'o' followed by return (which you cannot see). The second use of `cat` prints the file to the terminal as ASCII characters as one might expect. However, using the `od` command we can inspect the numerical representation. In particular using the option `-t` with format `u1` instructs `od` to format the content as a sequence of unsigned decimal integers in the range $0 \ldots 255$. As a result we see a total of six numbers, namely 104, 101, 108, 108 and 111 representing characters 'h', 'e', 'l', 'l' and 'o' as in MEM, and the number 10 which represents the character *LF* (which occurs as a result of our pressing return after "hello" when creating `A.txt`).

1.2 Problem #2: representing strings

The second problem is that so far we do not really have a way to know how long a string is, i.e., how many characters it contains. What we have been doing is looking at the memory content

$$\begin{array}{rcccccccccc} i & = & & \ldots, & 3, & 4, & 5, & 6, & 7, & \ldots & \\ \mathsf{MEM} & = & \langle & \ldots, & 104, & 101, & 108, & 108, & 111, & \ldots & \rangle \\ \textsc{Chr}(\mathsf{MEM}) & = & & \ldots, & \text{`h'}, & \text{`e'}, & \text{`l'}, & \text{`l'}, & \text{`o'}, & \ldots & \end{array}$$

then seeing the characters 'h', 'e', 'l', 'l' and 'o' and assuming that they are the string "hello". But how do we know the length of the string starting at address #3? It could be possible that addresses higher than #7 hold more characters so that we actually have a longer string than we thought. For example, maybe $\mathsf{MEM} = 46$ which means there is a full stop character there:

$$\begin{array}{rcccccccccc} i & = & & \ldots, & 3, & 4, & 5, & 6, & 7, & 8, & \ldots \\ \mathsf{MEM} & = & \langle & \ldots, & 104, & 101, & 108, & 108, & 111, & 46, & \ldots & \rangle \\ \textsc{Chr}(\mathsf{MEM}) & = & & \ldots, & \text{`h'}, & \text{`e'}, & \text{`l'}, & \text{`l'}, & \text{`o'}, & \text{`.'}, & \ldots \end{array}$$

How do we know if the full stop is actually *part* of the string starting at address #3, or if it is just some unrelated value which is there by chance? That is, how do we know we really mean the five character string "hello" rather than the six character string "hello."?

Given some address x which tells us where in memory a string starts, our goal is to add structure to the data so we know the length (and therefore what the intended content is). One way is to form a data structure by embedding some extra information in MEM to tell us how long the string is; there are at least two schemes we could consider:

1. We could embed the string length, which is a number, as the first element:

$$\begin{array}{rcccccccccc} i & = & & \ldots, & 3, & 4, & 5, & 6, & 7, & 8, & \ldots \\ \mathsf{MEM} & = & \langle & \ldots, & 5, & 104, & 101, & 108, & 108, & 111, & \ldots & \rangle \\ \textsc{Chr}(\mathsf{MEM}) & = & & \ldots, & \mathit{ENQ}, & \text{`h'}, & \text{`e'}, & \text{`l'}, & \text{`l'}, & \text{`o'}, & \ldots \end{array}$$

   We still have a string starting at address #3 but instead of interpreting the first element as a character, we interpret it as the length of the string. Since $\mathsf{MEM} = 5$ we know this string has five characters in it, and that they will be in addresses $4 \ldots 8$.

2. We could embed a character, which will mark the end of the string, as the last element:

$$\begin{array}{rcccccccccc} i & = & & \ldots, & 3, & 4, & 5, & 6, & 7, & 8, & \ldots \\ \mathsf{MEM} & = & \langle & \ldots, & 104, & 101, & 108, & 108, & 111, & 0, & \ldots & \rangle \\ \textsc{Chr}(\mathsf{MEM}) & = & & \ldots, & \text{`h'}, & \text{`e'}, & \text{`l'}, & \text{`l'}, & \text{`o'}, & \mathit{NUL}, & \ldots \end{array}$$

   Again we have a string starting at address #3, but now we can be sure where it ends because $\mathsf{MEM} = 0$. This is not a character as such, but the end of string marker or **terminator** which enables us to determine that the length is $8 - 3 = 5$ characters.

The reason these two schemes are interesting is because they are adopted by the Pascal [6] and C [5] programming languages respectively; because they do not really have good names[1] we will refer to them as the P-string and C-string schemes. There is no G-string or equivalent before you ask.

Neither the P-string or C-string scheme is right or wrong, but *do* give particular advantages and disadvantages in each case:

[1] Well, that is not *quite* true: the first option is often termed a length-prefixed string, and the second ASCIIZ which can be read as zero-terminated (or null-terminated) string.

P-string:

- The string is represented using one more element than the content suggests; if there are n characters, we need $n+1$ elements because one is used to store the length.
- Since the length needs to fit into one element of MEM, there is an upper limit on the lengths of string we can represent. We can remove this restriction by using more elements, but things become more tricky and we use more space.
- We have direct access to the length of the string, since we simply have to load it from the first element.

C-string:

- The string is represented using one more element than the content suggests; if there are n characters, we need $n+1$ elements because one is used to store the *NUL* terminator.
- The actual string content cannot contain a *NUL* character, otherwise we would interpret this as the terminator, i.e., the end of the string. Although *NUL* is not printable, the fact we cannot use it can be a disadvantage in some circumstances.
- There is no real restriction on the string length, but we do not have direct access to that length. To find the length of a string, we need to search for the NUL terminator.

These are just two data structures that could be used to represent strings. Still more exist: can you think of at least one more possibility? Explain how this third data structure works using an example, and contrast it with the C-string and P-string approaches by constructing a list of advantages and disadvantages.

2 String algorithms

The idea of introducing these two data structures is to demonstrate some more algorithms; unlike the algorithms in Chapter 3 that computed an arithmetic result, these ones operate on and manipulate instances of the string data structures. The real goal is to point out something subtle about the algorithms and data structures involved, but it will take a while to get there.

Keep in mind that we can reuse the ideas we developed in Chapter 3 to compare algorithms against each other. In this case, a good definition of the problem size n is the length of the strings are dealing with.

2.1 `strlen`: finding the length of a string

About the most simple task one might think of in the context of strings is computing their length. Given a string starting at address x, the idea is to return the number of characters in that string. The C programming language includes a standard function called `strlen` which represents an implementation of this idea.

2.1.1 P-string version

The P-string version is shown in Figure 3a. Imagine the memory content is initially

i	=	$\ldots,$	3,	4,	5,	6,	7,	8,	$\ldots$	
MEM	=	$\langle$ $\ldots,$	5,	104,	101,	108,	108,	111,	$\ldots$	$\rangle$
Chr(MEM)	=	$\ldots,$	*ENQ*,	'h',	'e',	'l',	'l',	'o',	$\ldots$	

and we invoke the algorithm using P-String-Length(3). The only input is the string address, so in this case we want to find the length of a string at address 3; the corresponding steps performed by the algorithm are as follows:

Step #1 Return MEM = 5.

This could not be simpler! Remember the string length is embedded as the first element, so the algorithm computes the required length in one step by simply loading it. Because of this, the algorithm *always* takes just one step; describing it using big-O notation we say it is $O(1)$ because no matter what the input size is, it takes a constant number of steps to give us a result.

```
1 algorithm P-String-Length(x) begin
2 |  return MEM
3 end
```

(a) *P-string version.*

```
1 algorithm C-String-Length(x) begin
2 |  i ← 0
3 |  while MEM ≠ 0 do
4 |  |  i ← i + 1
5 |  end
6 |  return i
7 end
```

(b) *C-string version.*

Figure 3: *Algorithms to compute the length of a string at address x, represented using a P-string (left-hand side) or C-string (right-hand side) data structure.*

2.1.2 C-string version

The C-string version is shown in Figure 3b. Imagine the memory content is initially

i	=		...,	3,	4,	5,	6,	7,	8,	...	
MEM	=	⟨	...,	104,	101,	108,	108,	111,	0,	...	⟩
Chr(MEM)	=		...,	'h',	'e',	'l',	'l',	'o',	*NUL*,	...	

which is just the C-string version of the P-string one we used above. If we invoke the algorithm with C-String-Length(3) in a similar way, the corresponding steps performed by the algorithm are now:

Step #1 Assign $i \leftarrow 0$.

Step #2 Since MEM $\neq 0$, assign $i \leftarrow i + 1$, i.e., $i \leftarrow 1$.

Step #3 Since MEM $\neq 0$, assign $i \leftarrow i + 1$, i.e., $i \leftarrow 2$.

Step #4 Since MEM $\neq 0$, assign $i \leftarrow i + 1$, i.e., $i \leftarrow 3$.

Step #5 Since MEM $\neq 0$, assign $i \leftarrow i + 1$, i.e., $i \leftarrow 4$.

Step #6 Since MEM $\neq 0$, assign $i \leftarrow i + 1$, i.e., $i \leftarrow 5$.

Step #7 Return $i = 5$.

This time things are a bit more complicated. So what is going on? We have used a slightly different loop construct than in Chapter 3 for this algorithm. When we write **while** X **do** Y

, the idea is that we repeatedly process the block Y until X evaluates to **false**. So we perform a test: if X evaluates to **true** then we process the block Y, and then start again, otherwise we do not bother and exit the loop. This sort of loop is **unbounded** because we do not know how many times we will process Y before we start.

Here, the test X is "have we found the string terminator". We keep adding one to a counter called i until MEM $= 0$ at which point we know that i should give the length of the string. So how many steps does the algorithm take? The answer is that if we add one to i for each character in the string, then we must take at least n steps for an n-character string. Using big-O notation we say it is $O(n)$.

So the P-string approach is much better when we want to compute the length of a string: an algorithm which has complexity $O(1)$ will always be better than one which has complexity $O(n)$, assuming n is large enough. In this case we do not even need n to be large since for *all* strings the P-string method will be better: there is only one string for which the two methods will take exactly the same number of steps. As an exercise, can you work out which string this is?

2.2 `toupper`: converting a string to upper-case

Computing the length of a string *accesses* the string content but does not *alter* it. The next task we look at is altering the string starting at address x so that the content consists of upper-case characters only: we take each lower-case character in the string and convert it into the upper-case equivalent. The C programming language does not include a standard function which represents quite this idea, but it does have a function called `toupper` that turns lower-case characters (rather than strings) into the upper-case equivalent.

2.2.1 P-string version

The P-string version is shown in Figure 4a. Imagine the memory content is initially

$$\begin{array}{rcccccccccc} i & = & \ldots, & 3, & 4, & 5, & 6, & 7, & 8, & \ldots & \\ \mathsf{MEM} & = & \langle \ \ldots, & 5, & 104, & 101, & 108, & 108, & 111, & \ldots & \rangle \\ \textsc{Chr}(\mathsf{MEM}) & = & \ldots, & \mathit{ENQ}, & \text{'h'}, & \text{'e'}, & \text{'l'}, & \text{'l'}, & \text{'o'}, & \ldots & \end{array}$$

and we invoke the algorithm using P-String-ToUpper(3). The only input is the string address, so in this case we want to convert the string at address 3 into upper-case. The corresponding steps performed by the algorithm are

Step # 1 Assign n = P-String-Length(3) = 5.

Step # 2 Assign $t = \mathsf{MEM} = 104$.

Step # 3 Since $97 \leq t \leq 122$, store the result, i.e., $\mathsf{MEM} = 104 - 32 = 72$.

Step # 4 Assign $t = \mathsf{MEM} = 101$.

Step # 5 Since $97 \leq t \leq 122$, store the result, i.e., $\mathsf{MEM} = 101 - 32 = 69$.

Step # 6 Assign $t = \mathsf{MEM} = 108$.

Step # 7 Since $97 \leq t \leq 122$, store the result, i.e., $\mathsf{MEM} = 108 - 32 = 76$.

Step # 8 Assign $t = \mathsf{MEM} = 108$.

Step # 9 Since $97 \leq t \leq 122$, store the result, i.e., $\mathsf{MEM} = 108 - 32 = 76$.

Step #10 Assign $t = \mathsf{MEM} = 111$.

Step #11 Since $97 \leq t \leq 122$, store the result, i.e., $\mathsf{MEM} = 111 - 32 = 79$.

Step #12 Return.

Of course this time the algorithm actually stores new content in memory rather than just reading existing content from it; after the algorithm has finished, the content is described by:

$$\begin{array}{rcccccccccc} i & = & \ldots, & 3, & 4, & 5, & 6, & 7, & 8, & \ldots & \\ \mathsf{MEM} & = & \langle \ \ldots, & 5, & 72, & 69, & 76, & 76, & 79, & \ldots & \rangle \\ \textsc{Chr}(\mathsf{MEM}) & = & \ldots, & \mathit{ENQ}, & \text{'H'}, & \text{'E'}, & \text{'L'}, & \text{'L'}, & \text{'O'}, & \ldots & \end{array}$$

How did it end up this way? In the example where we were finding the length of a string, the complication was a new type of loop to deal with; here the complication is that by invoking P-String-ToUpper, we invoke P-String-Length as well during the first step. From then on, it is fairly easy to see what is going on. We perform a series of loads from memory to retrieve the "current" character at MEM, and if the character we load is in the range 97 . . . 122 (i.e., it is a lower-case character) we subtract 32 from it and store it back at the same place.

Working out how to express this in big-O notation is not hard. The main loop in lines #3 to #8 clearly takes $2 \cdot n$ steps since, for each character, we need one step to load it and one step to test and assign it if applicable. But to be fair, we *also* need to include the number of steps that it took to compute the string length via P-String-Length. In a sense we just add the number of steps together, so informally we might write $O(1) + O(n) = O(1 + n)$. Of course $O(n)$ dominates $O(1)$ so using the same sort of simplification we did in Chapter 3, this turns into $O(n)$.

2.2.2 C-string version

The C-string version is shown in Figure 4b. Imagine the memory content is initially

$$\begin{array}{rcccccccccc} i & = & \ldots, & 3, & 4, & 5, & 6, & 7, & 8, & \ldots & \\ \mathsf{MEM} & = & \langle \ \ldots, & 104, & 101, & 108, & 108, & 111, & 0, & \ldots & \rangle \\ \textsc{Chr}(\mathsf{MEM}) & = & \ldots, & \text{'h'}, & \text{'e'}, & \text{'l'}, & \text{'l'}, & \text{'o'}, & \mathit{NUL}, & \ldots & \end{array}$$

which, again, is just the C-string version of the P-string one we used above. If we invoke the algorithm with C-String-ToUpper(3) again, the corresponding steps performed by the algorithm are

Step # 1 Assign n = C-String-Length(3) = 5.

```
1  algorithm P-String-ToUpper(x) begin
2  |  n ← P-String-Length(x)
3  |  for i from 0 upto n − 1 do
4  |  |  t ← MEM
5  |  |  if 97 ≤ t ≤ 122 then
6  |  |  |  MEM ← t − 32
7  |  |  end
8  |  end
9  |  return
10 end
```

(a) *P-string version.*

```
1  algorithm C-String-ToUpper(x) begin
2  |  n ← C-String-Length(x)
3  |  for i from 0 upto n − 1 do
4  |  |  t ← MEM
5  |  |  if 97 ≤ t ≤ 122 then
6  |  |  |  MEM ← t − 32
7  |  |  end
8  |  end
9  |  return
10 end
```

(b) *C-string version.*

Figure 4: *Algorithms to convert a string at address x, represented using a P-string (left-hand side) or C-string (right-hand side) data structure, into upper-case.*

Step # 2 Assign t = MEM = 104.

Step # 3 Since $97 \leq t \leq 122$, store the result, i.e., MEM = $104 - 32 = 72$.

Step # 4 Assign t = MEM = 101.

Step # 5 Since $97 \leq t \leq 122$, store the result, i.e., MEM = $101 - 32 = 69$.

Step # 6 Assign t = MEM = 108.

Step # 7 Since $97 \leq t \leq 122$, store the result, i.e., MEM = $108 - 32 = 76$.

Step # 8 Assign t = MEM = 108.

Step # 9 Since $97 \leq t \leq 122$, store the result, i.e., MEM = $108 - 32 = 76$.

Step #10 Assign t = MEM = 111.

Step #11 Since $97 \leq t \leq 122$, store the result, i.e., MEM = $111 - 32 = 79$.

Step #12 Return.

which alter the memory content to read

i	=	...,	3,	4,	5,	6,	7,	8,	...	
MEM	=	⟨ ...,	72,	69,	76,	76,	79,	0,	...	⟩
Chr(MEM)	=	...,	'H',	'E',	'L',	'L',	'O',	*NUL*,	...	

afterwards. In the example where we were finding the length of a string, the steps taken by the two algorithms were radically different; here they are more similar. They are more or less the same in fact, bar the first step which obviously needs to use C-String-Length instead of P-String-Length. Since C-String-Length takes $O(n)$ steps, the function C-String-ToUpper takes $O(n) + O(n) = O(2 \cdot n)$ steps. The constant can be ignored if we again tolerate the simplifications described in Chapter 3, and we end up with $O(n)$.

2.3 `strcmp`: testing if one string is the same as another

Next we look at the task of string comparison, or string matching. The idea is that we take two strings, one starting at address x and one at address y, and try to determine whether they are the same or not. The C programming language includes a standard function called `strcmp` which represents an implementation of this idea.

2.3.1 P-string version

The P-string version is shown in Figure 5a. Imagine the memory content is initially

i	=		…,	3,	4,	5,	6,	7,	8,	
MEM	=	⟨	…,	5,	104,	101,	108,	108,	111,	⟩
CHR(MEM)	=		…,	*ENQ*,	‘h’,	‘e’,	‘l’,	‘l’,	‘o’,	

i	=		9,	10,	11,	12,	13,	14,	…	
MEM	=	⟨	5,	104,	101,	76,	76,	111,	…	⟩
CHR(MEM)	=		*ENQ*,	‘h’,	‘e’,	‘L’,	‘L’,	‘o’,	…	

Notice that we have got some more content: in fact *so* much more we have now split the memory content onto two lines. This is, in part, because this algorithm takes two arguments representing the addresses of the strings we want to compare. Using P-STRING-MATCH(3, 9) to invoke the algorithm means we want to compare the strings at addresses 3 and 9, with the corresponding steps performed as follows:

Step # 1 Assign n = P-STRING-LENGTH(3) = 5.

Step # 2 Assign m = P-STRING-LENGTH(9) = 5.

Step # 3 Since $n = m$, continue.

Step # 4 Since MEM = MEM = 104, continue.

Step # 5 Since MEM = MEM = 101, continue.

Step # 6 Since MEM ≠ MEM, return **false**.

The strings are *similar*, but not the *same*. In particular, the ‘l’ characters in the string at address #3 (i.e., the characters at addresses #6 and #7) are lower-case, but those in the string at address #9 (i.e., the characters at addresses #12 and #13) are upper-case. This is reflected in the fact that the algorithm returned **false**.

To understand the reason it gives the right answer we need to take a closer look at what happens at each step. The first thing that happens is a condition, namely if the number of characters in the first string is not the same as the number of characters in the second string then the strings cannot be the same, and the algorithm returns **false** as the result. But that does not happen here, since both strings have five characters in them so we carry on, and proceed to check each character. To do this, we use a loop which iterates over a block for values of i in the range $0 \ldots n-1$. By this point we know $n = m$ so there is no danger of the i-th character of either string being “out of bounds” (i.e., beyond their actual length). For each value of i, we test the i-th characters of the two strings against each other. If they are not equal, then clearly the two strings are not equal and we can return **false** as the result. For $i = 0$ and $i = 1$, the characters in both strings are ‘h’ and ‘e’ so the algorithm continues. When we hit $i = 3$, however, the characters are ‘l’ and ‘L’ which are *not* equal; the algorithm notices this, concludes that the strings do not match, then returns **false** as the result. If, on the other hand, it had got all the way through the strings and found that they *all* matched it would return **true** instead.

As you might have expected, there is an extra complication here. The number of steps the algorithm takes depends on the strings themselves, i.e., their content and length, but it is not too hard to describe the number of steps in big-O notation. The two invocations of P-STRING-LENGTH take $O(1) + O(1)$ steps which is still $O(1)$, and then in the worst case we have to test all n characters of the strings so the main loop takes $O(n)$. The total number of steps is therefore $O(1) + O(n)$ which simplifies to $O(n)$.

2.3.2 C-string version

The C-string version is shown in Figure 5b. Imagine the memory content is initially

i	=		…,	3,	4,	5,	6,	7,	8,	
MEM	=	⟨	…,	104,	101,	108,	108,	111,	0,	⟩
CHR(MEM)	=		…,	‘h’,	‘e’,	‘l’,	‘l’,	‘o’,	*NUL*,	

i	=		9,	10,	11,	12,	13,	14,	…	
MEM	=	⟨	104,	101,	76,	76,	111,	0,	…	⟩
CHR(MEM)	=		‘h’,	‘e’,	‘L’,	‘L’,	‘o’,	*NUL*,	…	

which, again, is just the C-string version of the P-string one we used above. Invoking the algorithm with C-STRING-MATCH(3, 9) gives basically the same steps as last time, namely

Step # 1 Assign n = C-STRING-LENGTH(3) = 5.

```
1  algorithm P-String-Match(x, y) begin
2  |  n ← P-String-Length(x)
3  |  m ← P-String-Length(y)
4  |  if n ≠ m then
5  |  |  return false
6  |  end
7  |  for i from 0 upto n − 1 do
8  |  |  if MEM ≠ MEM then
9  |  |  |  return false
10 |  |  end
11 |  end
12 |  return true
13 end
```

(a) *P-string version.*

```
1  algorithm C-String-Match(x, y) begin
2  |  n ← C-String-Length(x)
3  |  m ← C-String-Length(y)
4  |  if n ≠ m then
5  |  |  return false
6  |  end
7  |  for i from 0 upto n − 1 do
8  |  |  if MEM ≠ MEM then
9  |  |  |  return false
10 |  |  end
11 |  end
12 |  return true
13 end
```

(b) *C-string version.*

Figure 5: *Algorithms to match one string at address x against another at address y, both represented using a P-string (left-hand side) or C-string (right-hand side) data structure.*

Step # 2 Assign m = C-String-Length(9) = 5.

Step # 3 Since $n = m$, continue.

Step # 4 Since MEM = MEM = 104, continue.

Step # 5 Since MEM = MEM = 101, continue.

Step # 6 Since MEM ≠ MEM, return **false**.

This time we use C-String-Length instead of P-String-Length, but the result is still **false** as you would expect. The two initial invocations of C-String-Length plus the main algorithm take a grand total of $O(n)+O(n)+O(n) = O(3 \cdot n)$ steps, but of course we again ignore the constant and simplify this to $O(n)$.

2.4 `strcat`: concatenating two strings together

The final task we look at is the key one we have been building up to, as you might have guessed. The idea is to take one string (the source string) starting at address y and concatenate it with, or join it onto, another one (the target string) starting at address x; the source string remains unaltered.

2.4.1 P-string version

The P-string version is shown in Figure 6a. Imagine the memory content is initially

i	=		…,	3,	4,	5,	6,	7,	8,	
MEM	=	⟨	…,	5,	104,	101,	108,	108,	111,	⟩
Chr(MEM)	=		…,	*ENQ*,	‘h’,	‘e’,	‘l’,	‘l’,	‘o’,	

i	=			9,	10,	11,	12,	13,	14,	…
MEM	=	⟨		5,	104,	101,	108,	108,	111,	… ⟩
Chr(MEM)	=			*ENQ*,	‘h’,	‘e’,	‘l’,	‘l’,	‘o’,	…

and we invoke the algorithm using P-String-Concat(9, 3). This means we specify that the target string starts at address #9, and the source string starts at address #3; both strings are initially “hello”. The algorithm itself is quite simple. Basically, after computing the lengths of the source and target string, it uses a loop to copy each character of the source string to the corresponding address *after* the target string ends. In short, the character at address $y + 1 + i$ in the source string is copied to address $x + 1 + i + n$ in the target string. The steps performed by the algorithm in this case are

Step # 1 Assign n = P-String-Length(9) = 5.

Step # 2 Assign m = P-String-Length(3) = 5.

Step # 3 Assign MEM = MEM = 104.

```
1 algorithm P-String-Concat(x, y) begin
2 |   n ← P-String-Length(x)
3 |   m ← P-String-Length(y)
4 |   for i from 0 upto m − 1 do
5 |   |   MEM ← MEM
6 |   end
7 |   MEM ← n + m
8 |   return
9 end
```

(a) *P-string version.*

```
1 algorithm C-String-Concat(x, y) begin
2 |   n ← C-String-Length(x)
3 |   m ← C-String-Length(y)
4 |   for i from 0 upto m − 1 do
5 |   |   MEM ← MEM
6 |   end
7 |   MEM ← 0
8 |   return
9 end
```

(b) *C-string version.*

Figure 6: *Algorithms to concatenate (i.e., join) one string at address y onto the end of another at address x, both represented using a P-string (left-hand side) or C-string (right-hand side) data structure.*

Step # 4 Assign MEM = MEM = 101.

Step # 5 Assign MEM = MEM = 108.

Step # 6 Assign MEM = MEM = 108.

Step # 7 Assign MEM = MEM = 111.

Step # 8 Assign MEM = 5 + 5 = 10.

Step # 9 Return.

which alter the memory content to read

$$
\begin{array}{rcccccccccc}
i & = & \ldots, & 3, & 4, & 5, & 6, & 7, & 8, & & \\
\mathsf{MEM} & = & \langle \ \ldots, & 5, & 104, & 101, & 108, & 108, & 111, & & \rangle \\
\textsc{Chr}(\mathsf{MEM}) & = & \ldots, & \mathit{ENQ}, & \text{'h'}, & \text{'e'}, & \text{'l'}, & \text{'l'}, & \text{'o'}, & &
\end{array}
$$

$$
\begin{array}{rcccccccccc}
i & = & & 9, & 10, & 11, & 12, & 13, & 14, & & \\
\mathsf{MEM} & = & \langle & 10, & 104, & 101, & 108, & 108, & 111, & & \rangle \\
\textsc{Chr}(\mathsf{MEM}) & = & & \mathit{LF}, & \text{'h'}, & \text{'e'}, & \text{'l'}, & \text{'l'}, & \text{'o'}, & &
\end{array}
$$

$$
\begin{array}{rcccccccccc}
i & = & & 15, & 16, & 17, & 18, & 19, & \ldots & & \\
\mathsf{MEM} & = & \langle & 104, & 101, & 108, & 108, & 111, & \ldots & & \rangle \\
\textsc{Chr}(\mathsf{MEM}) & = & & \text{'h'}, & \text{'e'}, & \text{'l'}, & \text{'l'}, & \text{'o'}, & \ldots & &
\end{array}
$$

We have split the content across three lines now to make it fit, but the point is that if you check the addresses and the content you find that the source string at address #3 is still "hello" but the target string at address #9 is now "hellohello", i.e., the original source and target joined together.

In terms of a big-O description of P-String-Concat, the two invocations of P-String-Length take $O(1)+O(1)$ steps which is still $O(1)$, and then we perform n steps inside the loop that copies the characters. This gives a total of $O(1) + O(1) + O(n)$ which of course we simplify to $O(n)$.

2.4.2 C-string version

The C-string version is shown in Figure 6b. Imagine the memory content is initially

$$
\begin{array}{rcccccccccc}
i & = & \ldots, & 3, & 4, & 5, & 6, & 7, & 8, & & \\
\mathsf{MEM} & = & \langle \ \ldots, & 104, & 101, & 108, & 108, & 111 & 0, & & \rangle \\
\textsc{Chr}(\mathsf{MEM}) & = & \ldots, & \text{'h'}, & \text{'e'}, & \text{'l'}, & \text{'l'}, & \text{'o'} & \mathit{NUL}, & &
\end{array}
$$

$$
\begin{array}{rcccccccccc}
i & = & & 9, & 10, & 11, & 12, & 13, & 14, & \ldots & \\
\mathsf{MEM} & = & \langle & 104, & 101, & 108, & 108, & 111, & 0, & \ldots & \rangle \\
\textsc{Chr}(\mathsf{MEM}) & = & & \text{'h'}, & \text{'e'}, & \text{'l'}, & \text{'l'}, & \text{'o'}, & \mathit{NUL}, & \ldots &
\end{array}
$$

which, again, is just the C-string version of the P-string one we used above. Invoking the algorithm with C-String-Concat(9, 3) gives basically the same steps as last time, i.e.,

Step # 1 Assign n = C-String-Length(9) = 5.

Step # 2 Assign m = C-String-Length(3) = 5.

Step # 3 Assign MEM = MEM3 + 0 = 104.

Step # 4 Assign MEM = MEM3 + 1 = 101.

Step # 5 Assign MEM = MEM3 + 2 = 108.

Step # 6 Assign MEM = MEM3 + 3 = 108.

Step # 7 Assign MEM = MEM3 + 4 = 111.

Step # 8 Assign MEM = 0.

Step # 9 Return.

The only real difference is the last step, which instead of setting the length of the target string to $n + m$, i.e., the sum of the lengths of the source and target, it "moves" the terminator character to the correct position at the new end of the target. The memory content is altered to read

i	=		...,	3,	4,	5,	6,	7,	8,	
MEM	=	⟨	...,	104,	101,	108,	108,	111	0,	⟩
Chr(MEM)	=		...,	'h',	'e',	'l',	'l',	'o'	*NUL*,	

i	=			9,	10,	11,	12,	13,	14,	
MEM	=	⟨		104,	101,	108,	108,	111,	104,	⟩
Chr(MEM)	=			'h',	'e',	'l',	'l',	'o',	'h',	

i	=			15,	16,	17,	18,	19,	...	
MEM	=	⟨		101,	108,	108,	111,	0,	...	⟩
Chr(MEM)	=			'e',	'l',	'l',	'o',	*NUL*,	...	

so that again we find the source string at address #3 is still "hello" but the target string at address #9 is now "hellohello", i.e., the original source and target joined together. Again, the big-O notation for C-String-Concat is simple. We end up with $O(n) + O(n) + O(n)$ which we simplify to $O(n)$.

You can probably think of *lots* of other useful operations we could perform on a string, but two examples are described below. In each case, your challenge is to write two algorithms that apply the operation using the C-string and P-string data structures respectively.

1. C has a standard function called `strchr` that takes a character c and the address of a string x as input: the function returns how far the first instance of the character is (i.e., the offset) from the start of the string. For example, if the character is 'l' and the string is "hello" we expect the result to be 2: the first instance of 'l' is 2 characters from the start of "hello".

   What happens if the character does not occur in the string: what useful value could the algorithm return in this case?

2. C has no standard function called `strrev`, but imagine it takes the address of a string x as input and reverses the order of the characters. For example, the string "hello" would become "olleh".

2.5 Problem #3: repeated concatenation

After all that, here is the problem that justifies what we have been doing. Have a look again at the last task of concatenating strings together and imagine we try to concatenate m strings together, accumulating the result in the string at address x. Using P-String-Concat as an example, we would end up with something like this:

P-String-Concat(x, "foo") $\mapsto$ x = "foo"
P-String-Concat(x, "bar") $\mapsto$ x = "foobar"
P-String-Concat(x, "baz") $\mapsto$ x = "foobarbaz"

That is, after the first invocation we would have appended "foo" to the empty string to get "foo", and after the second we would have appended "bar" to "foo" to get "foobar" and so on.

The thing to notice is that x gets longer and longer, but each string we concatenate to it remains short (say n characters). So how does this alter the number of steps taken by each invocation of P-String-Concat? Well, the number of steps in the main loop is determined by the number of characters in the source string, so this does not change. What about the number of steps required *before* the main loop to compute the length of the strings? P-String-Length is $O(1)$, so no matter how long x is it will take a constant number of steps to give us a result. Even if we invoke P-String-Length m times like this, the end result is still $O(n)$.

What about the C-string alternative? Again starting off with x as the empty string, we would get the same result, i.e.,

$$\begin{array}{lcl} \text{C-String-Concat}(x, \text{“foo”}) & \mapsto & x = \text{“foo”} \\ \text{C-String-Concat}(x, \text{“bar”}) & \mapsto & x = \text{“foobar”} \\ \text{C-String-Concat}(x, \text{“baz”}) & \mapsto & x = \text{“foobarbaz”} \end{array}$$

but now the behaviour is a bit different. Of course, the number of steps is still determined by the number of characters in the source string and this still does not change. But now, as x gets longer and longer C-String-Length takes more and more steps to give us a result. Remember, it is $O(n)$ and n is getting larger and larger with each invocation. This might not look that bad, because if we invoke C-String-Concat four times then the total number of steps will be

$$O(n) + O(2n) + O(3n) + O(4n) = O(10n),$$

but we simplify this by ignoring the constants to get $O(n)$. But what happens if we invoke C-String-Concat m times? Now we get the sum

$$O(n) + O(2n) + \cdots + O(m \cdot n) = O\left(\left(\sum_{i=1}^{m} i\right) \cdot n\right)$$

which simplifies first to

$$O\left((m \cdot (m+1)/2) \cdot n\right)$$

because $\sum_{i=1}^{m} i = m(m+1)/2$ and then to

$$O(n \cdot m^2)$$

because $O(m(m+1)/2) = O(m^2)$. So the upshot is that if we invoke C-String-Concat m times like this, the end result is more like $O(m^2)$, since now we treat the value n as the constant. Yikes! Remember Chapter 3? $O(m^2)$ is *bad*. An eloquent analogy of this problem is offered by famed writer and programmer Joel Spolsky [8]:

> *Shlemiel gets a job as a street painter, painting the dotted lines down the middle of the road. On the first day he takes a can of paint out to the road and finishes 300 yards of the road. "That's pretty good!" says his boss, "you're a fast worker!" and pays him a kopeck. The next day Shlemiel only gets 150 yards done. "Well, that's not nearly as good as yesterday, but you're still a fast worker. 150 yards is respectable," and pays him a kopeck. The next day Shlemiel paints 30 yards of the road. "Only 30!" shouts his boss. "That's unacceptable! On the first day you did ten times that much work! What's going on?" "I can't help it," says Shlemiel. "Every day I get farther and farther away from the paint can!"*

The C-string `strcat` algorithm *is* poor old Shlemiel in this analogy: each time we try to concatenate one of our m strings, we need to place it further and further away from the end of the start of x.

Hopefully you can appreciate the problem now that we have spelled it out in such gruesome detail, but why does it *matter*? In short, understanding problems like this highlights the fact that C hides low-level detail of the data structure from us; this is even more true of languages such as Java. On one hand the abstraction this offers is great news because we do not have to worry about the data structure so much: the programming language takes care of it all for us automatically. But on the other hand, *only* by understanding low-level details can one hope to write efficient high-level programs! This is an issue that, in the opinion of many people, plagues modern Computer Science. We have built great tools to abstract away detail so we can construct wondrous hardware and software artefacts, but without an understanding of the fundamentals, one is *always* at a disadvantage.

C is not a bad language, and the C-string data structure is not the wrong choice: remember each of the C-string and P-string approaches have advantages and disadvantages. Therefore, it is interesting to look at the reason *why* the designers of C, Brian Kernighan and Dennis Ritchie, chose C-string rather than the P-string alternative in the 1970*s*. Two historical drivers are important. First, memory was at a real premium at the time C was developed; when dealing with large strings, the advantage of having a single string terminator character rather than a larger length field is tangible. It might seem amazing now, but saving even an extra byte of memory here and there could have been important then. Second, the PDP [7] range of computers used as early development platforms for C already used C-string type strings, and had some instructions that could deal with such strings quite efficiently. Hindsight is a wonderful thing; maybe they would have made a different choice looking back, maybe not. But it should be clear that by understanding low-level details, one at least has the opportunity to learn from the benefit of hindsight, and potentially to design better data structures

and algorithms as a result. After all, data structures get *much* more complicated than strings so the cost of not understanding the details is potentially *much* greater as well.

Lots of instruction sets include instructions that were once useful, but now seem slightly odd; these legacy instructions often remain to ensure **backward compatibility**, i.e., to make sure old programs still work.

The x86 instruction set used by Intel includes support for something called **Binary Coded Decimal (BCD)** [3]. Find out about the instructions available for BCD; what do you think the original motivation for including them was? Why do you think they are or are not still useful now?

References

[1] *Wikipedia: ASCII*. http://en.wikipedia.org/wiki/ASCII (see p. 4).

[2] *Wikipedia: ASCII art*. http://en.wikipedia.org/wiki/ASCII_art (see p. 6).

[3] *Wikipedia: Binary Coded Decimal (BCD)*. http://en.wikipedia.org/wiki/Binary-coded_decimal (see p. 17).

[4] *Wikipedia: Bulletin Board System (BBS)*. http://en.wikipedia.org/wiki/Bulletin_board_system (see p. 6).

[5] *Wikipedia: C*. http://en.wikipedia.org/wiki/C_(programming_language) (see p. 7).

[6] *Wikipedia: Pascal*. http://en.wikipedia.org/wiki/Pascal_(programming_language) (see p. 7).

[7] *Wikipedia: Programmed Data Processor (PDP)*. http://en.wikipedia.org/wiki/Programmed_Data_Processor (see p. 16).

[8] *Wikipedia: "Schlemiel the painter's" algorithm*. http://en.wikipedia.org/wiki/Schlemiel_the_painter's_Algorithm (see p. 16).

I have a question/comment/complaint for you. Any (positive *or* negative) feedback, experience or comment is very welcome; this helps us to improve and extend the material in the most useful way.

However, keep in mind we are far from perfect; mistakes are of course possible, although hopefully rare. Some cases are hard for us to check, and make your feedback even more valuable: for instance

1. minor variation in software versions can produce subtle differences in how some commands and hence examples work, and
2. some examples download and use online resources, but web-sites change over time (or even might differ depending on where you access them from) so might cause the example to fail.

Either way, if you spot a problem then let us know: we will try to explain and/or fix things as fast as we can!

Can I use this material for something ? We are distributing these PDFs under the Creative Commons Attribution-ShareAlike License (CC BY-SA)

http://creativecommons.org/licenses/by-sa/4.0/

This is a fancy way to say you can basically do whatever you want with them (i.e., use, copy and distribute it however you see fit) as long as a) you correctly attribute the original source, and b) you distribute the result under the same license.

Is there a printed version of this material I can buy? Yes: Springer have published selected Chapters in

http://www.springer.com/computer/book/978-3-319-04041-7

while *also* allowing us to keep electronic versions online. Any royalties from this published version are donated to the UK-based Computing At School (CAS) group, whose ongoing work can be followed at

http://www.computingatschool.org.uk/

Why are all your references to Wikipedia? Our goal is to give an easily accessible overview, so it made no sense to reference lots of research papers. There are basically two reasons why: research papers are often written in a way that makes them hard to read (even when their intellectual content is not difficult to understand), and although many research papers are available on the Internet, many are not (or have to be paid for). So although some valid criticisms of Wikipedia exist, for introductory material on Computer Science it certainly represents a good place to start.

I like programming; why do the examples include so little programming? We want to focus on interesting topics rather than the mechanics of programming. So even when we include example programs, the idea is to do so in a way where their meaning is fairly clear. For example it makes more sense to use pseudo-code algorithms or reuse existing software tools than complicate a description of something by including pages and pages of program listings.

But you need to be able to program to do Computer Science, right? Yes! But only in the same way as you need to be able to read and write to study English. Put another way, reading and writing, or grammar and vocabulary, are just tools: they simply allow us to study topics such as English literature. Computer Science is the same. Although it *is* possible to study programming as a topic in itself, we are more interested in what can be achieved *using* programs: we treat programming itself as another tool.