

Introduction to Basic Algorithms

We have discussed many of the common data structures in Computer Science. Since this is also a course that discusses algorithms and algorithmic analysis, I wanted to spend the next couple of weeks discussing both basic and intermediate algorithms. We will finish the course with an MIT lecture about NP-completeness.

I am going to go over three basic-style algorithms we can use for practice in solving problems. I will give solutions, but you are welcome to practice on your own to create your own solutions. Solving problems (and knowing the correct data structure to use for the solution) is what Computer Science is all about.

The three algorithms we are going to look at are:

- Create a string tokenizer from scratch
- Writing a “toUpper” method from scratch (using ASCII / Unicode values)
- Finding the bit value of an integer at a given bit slot

String Tokenizer from scratch (algorithm)

The idea behind this algorithm is that we want to create a class that will work in the same way as Java's StringTokenizer class. (If you do not know about this class, research it!) We don't have to support all of the functionality of that class but what we do want to support are the following:

- Create a tokenizer object given a raw string and a delimiting character
- Support the "hasMoreTokens" method to determine if there are more tokens in the buffer
- Support the "nextToken" method to retrieve the next token in buffer
- Support the "countTokens" method as a way of passing the number of tokens left in buffer

This gives us the bare functionality of a string tokenizer. Here is the class documentation view of the class:

```
public class Tokenizer{  
    public Tokenizer(String raw, char delimiter); // Constructor  
  
    public boolean hasMoreTokens();                // Does the buffer have more tokens left?  
  
    public String nextToken();                      // Return next token in buffer or null  
  
    public int countTokens();                      // Number of tokens left in buffer  
  
}
```

The next page will discuss the implementation of this.

Tokenizer implementation

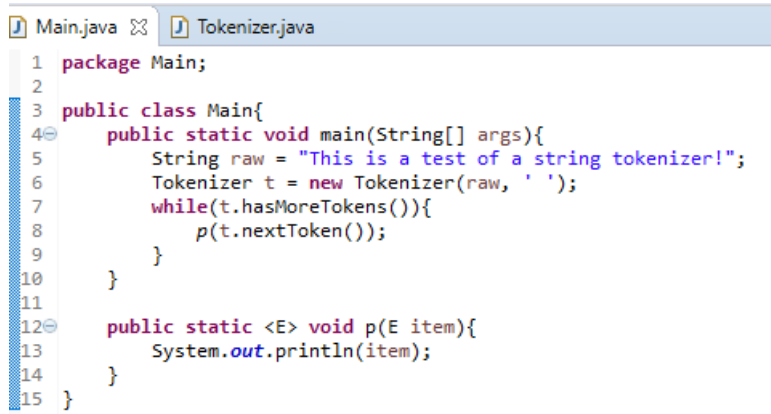
Before we implement this, the first question should be “what data structure should we use behind the scenes to hold the tokens?”. There is no right or wrong answer to this but I would recommend a Queue or an ArrayList. Since a Queue is more basic than an ArrayList, I am going to use a Queue to represent this.

Here is the code for my implementation. I encourage (and recommend) you to write your own implementation for algorithmic practice:

```
Main.java Tokenizer.java
1 package Main;
2
3 import java.util.LinkedList;
4 import java.util.Queue;
5
6 public class Tokenizer{
7     // Fields
8     private Queue<String> q;
9
10    // Constructor
11    public Tokenizer(String raw, char delimiter){
12        q = new LinkedList<String>();
13        init(raw, delimiter);
14    }
15
16    // Methods
17
18    /* This is an initialization workhorse method (parse the raw string into tokens) */
19    private void init(String raw, char delimiter){
20        int cursor = 0;
21        int i;
22        for(i = 0; i < raw.length(); i++){
23            if(raw.charAt(i) == delimiter){
24                // Prevent empty tokens (such as delimiter at beginning of string)
25                if(cursor == i){
26                    cursor = i + 1;
27                    continue;
28                }
29                String token = raw.substring(cursor, i); // substring end index is EXCLUSIVE!
30                q.add(token);
31                cursor = i + 1;
32            }
33        }
34        // Handle case where the raw string doesn't end on a delimiter and gather final token
35        if(raw.charAt(raw.length()-1) != delimiter){
36            String token = raw.substring(cursor, raw.length()); // substring end index is EXCLUSIVE!
37            q.add(token);
38        }
39    }
40
41    public boolean hasMoreTokens(){
42        return !q.isEmpty();
43    }
44
45    public String nextToken(){
46        return q.remove();
47    }
48
49    public int countTokens(){
50        return q.size();
51    }
52 }
```

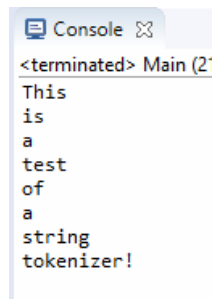
Tokenizer usage

Here is an example of how we can use the class we made in practice:



```
1 package Main;
2
3 public class Main{
4     public static void main(String[] args){
5         String raw = "This is a test of a string tokenizer!";
6         Tokenizer t = new Tokenizer(raw, ' ');
7         while(t.hasMoreTokens()){
8             p(t.nextToken());
9         }
10    }
11
12    public static <E> void p(E item){
13        System.out.println(item);
14    }
15 }
```

And here is the output to the console:



```
<terminated> Main (2'
This
is
a
test
of
a
string
tokenizer!
```

Asymptotic

This is an $O(n)$ time operation since we must traverse the whole string (char array).

The toUpper method from scratch (algorithm)

The idea behind this method is to evaluate a string, see if there are any a-z characters, and if so, convert those characters to upper case.

To do this properly, we need an understanding of the ASCII character set and how Java stores char variables. We might see the lower-case letter t as 't' but Java internally stores it as a Unicode (first 8 bits of Unicode are ASCII) value which is simply an index into the ASCII table.

Here is an ASCII table of the most common characters:

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
\$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
;	59	0073	0x3b	[91	0133	0x5b	{	123	0173	0x7b
<	60	0074	0x3c	\	92	0134	0x5c		124	0174	0x7c
=	61	0075	0x3d]	93	0135	0x5d	}	125	0175	0x7d
>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
?	63	0077	0x3f	_	95	0137	0x5f				

As you see, the character of 't' is 116 in base 10 (Dec on the chart for "decimal"). The character of 'T' is 84 in base 10. This means that the offset of upper case from lower case is $(84 - 116) = -32$. Therefore, if we find a character between 'a' and 'z', we can subtract 32 from that character to find the upper-case value.

Considerations

It is important to note that we do not subtract 32 from every character; only characters that are lower case 'a' through lower case 'z'. Otherwise, we will destroy the structure of the rest of the string. Numbers do not have cases in this table, so we ignore them. We also ignore the other symbols such as punctuation, etc.

The next page will discuss the implementation of this.

Implementation of toUpper

Since this method is basic in scope, we can make this a single static method; there is no need to create a class with fields to accommodate it.

Here is my implementation (although I encourage you to write your own version):

```
public static String toUpper(String raw){
    String ret = "";
    for(int i = 0; i < raw.length(); i++){
        char c = raw.charAt(i);
        if(c >= 'a' && c <= 'z'){
            c -= 32;
        }
        ret += c;
    }
    return ret;
}
```

Usage

This method will return the converted string.

Example:

```
String str = "this is a lower case string";

str = toUpper(str); // The string "str" will now be in all caps.
```

Asymptotic

This is an $O(n)$ time operation since we must traverse the whole string (char array).

Bit extraction (algorithm)

The idea behind this is that we want to find the value of any bit within an integer when given the position of the bit we are looking for.

To illustrate, let's take a number like 15. In binary, this is:

1 1 1 1

From right to left, we can number the “slots” of the bits using 0 to n-1. Therefore, we would say that bits 0, 1, 2, and 3 are set to 1. All higher numbered bits are set to 0. Therefore, if we have a 32-bit integer, we can say that the bits are numbered from 0 to 31.

Why would we need to do this?

Answer: On embedded systems, memory resources are often limited. Therefore, we may need to “bit pack” to save space. Another reason to do this is when working with hardware. Often times, hardware uses bit packing and extracting to conserve precious transistors within. Whatever the reason, as a Computer Scientist, you should know how to do this.

Anyhow, back to the algorithm. In Java, we can “mask” a bit to find out the true value by using the bitwise AND (&) and the value of 1. Consider this routine to find whether a number is odd or even:

```
int number = 31;
```

```
int oddBit = number & 1;    // This “masks” out the other bits to expose only the value of the odd bit
```

For our algorithm, we want to be able to return the bit value of a given “slot” within the integer. Here is the method prototype:

```
public static int getBit(int value, int bitNum);    // Return 0 or 1 for the bit slot in value
```

Bit extraction (implementation)

Here is my solution (I encourage you to write your own routine for practice):

```
public static int getBit(int value, int bitNum){
    int shift = 1 << bitNum;
    int ret = value & shift;
    ret = ret >> bitNum;
    return ret;
}
```

Another version for those who like short and concise code is:

```
public static int getBit2(int value, int bitNum){
    return (value & (1 << bitNum)) >> bitNum;
}
```

Usage

We can find the bit value (0/1) of a given bit as follows:

```
int bit = getBit(15, 3); // Find value of slot #3 (fourth bit from right)
```

```
// The value, in this case, will be a 1
```

Asymptotic

This is a constant time $O(1)$ operation since the number of bits in the variable do not affect the time to extract the value. Binary Shifting is a single machine code operation; not iterative.