Chapter 11

Strings

Strings have their own chapter in this text purely because string operations and transformations are incredibly frequent within programs. The algorithms presented are based on problems the authors have come across previously, or were formulated to satisfy curiosity.

11.1 Reversing the order of words in a sentence

Defining algorithms for primitive string operations is simple, e.g. extracting a sub-string of a string, however some algorithms that require more inventiveness can be a little more tricky.

The algorithm presented here does not simply reverse the characters in a string, rather it reverses the order of words within a string. This algorithm works on the principal that words are all delimited by white space, and using a few markers to define where words start and end we can easily reverse them.

```
1) algorithm ReverseWords(value)
2)     Pre:  value ≠ ∅, sb is a string buffer
3)     Post: the words in value have been reversed
4)     last ← value.Length − 1
5)     start ← last
6)     while last ≥ 0
7)         // skip whitespace
8)         while start ≥ 0 and value[start] = whitespace
9)             start ← start − 1
10)        end while
11)        last ← start
12)        // march down to the index before the beginning of the word
13)        while start ≥ 0 and start ≠ whitespace
14)            start ← start − 1
15)        end while
16)        // append chars from start + 1 to length + 1 to string buffer sb
17)        for i ← start + 1 to last
18)            sb.Append(value[i])
19)        end for
20)        // if this isn't the last word in the string add some whitespace after the word in the buffer
21)        if start > 0
22)            sb.Append(' ')
23)        end if
24)        last ← start − 1
25)        start ← last
26)    end while
27)    // check if we have added one too many whitespace to sb
28)    if sb[sb.Length −1] = whitespace
29)        // cut the whitespace
30)        sb.Length ← sb.Length −1
31)    end if
32)    return sb
33) end ReverseWords
```

11.2 Detecting a palindrome

Although not a frequent algorithm that will be applied in real-life scenarios detecting a palindrome is a fun, and as it turns out pretty trivial algorithm to design.

The algorithm that we present has a $O(n)$ run time complexity. Our algorithm uses two pointers at opposite ends of string we are checking is a palindrome or not. These pointers march in towards each other always checking that each character they point to is the same with respect to value. Figure 11.1 shows the *IsPalindrome* algorithm in operation on the string "Was it Eliot's toilet I saw?" If you remove all punctuation, and white space from the aforementioned string you will find that it is a valid palindrome.

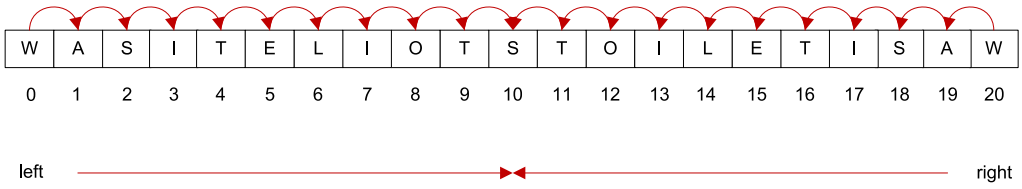

Figure 11.1: $left$ and $right$ pointers marching in towards one another

```
1) algorithm IsPalindrome(value)
2)     Pre:  value ≠ ∅
3)     Post:  value is determined to be a palindrome or not
4)     word ← value.Strip().ToUpperCase()
5)     left ← 0
6)     right ← word.Length −1
7)     while word[left] = word[right] and left < right
8)         left ← left + 1
9)         right ← right − 1
10)    end while
11)    return word[left] = word[right]
12) end IsPalindrome
```

In the *IsPalindrome* algorithm we call a method by the name of *Strip*. This algorithm discards punctuation in the string, including white space. As a result *word* contains a heavily compacted representation of the original string, each character of which is in its uppercase representation.

Palindromes discard white space, punctuation, and case making these changes allows us to design a simple algorithm while making our algorithm fairly robust with respect to the palindromes it will detect.

11.3 Counting the number of words in a string

Counting the number of words in a string can seem pretty trivial at first, however there are a few cases that we need to be aware of:

1. tracking when we are in a string
2. updating the word count at the correct place
3. skipping white space that delimits the words

As an example consider the string "Ben ate hay" Clearly this string contains three words, each of which distinguished via white space. All of the previously listed points can be managed by using three variables:

1. *index*
2. *wordCount*
3. *inWord*

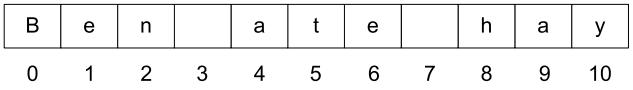

Figure 11.2: String with three words

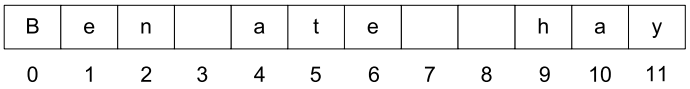

Figure 11.3: String with varying number of white space delimiting the words

Of the previously listed *index* keeps track of the current index we are at in the string, *wordCount* is an integer that keeps track of the number of words we have encountered, and finally $inWord$ is a Boolean flag that denotes whether or not at the present time we are within a word. If we are not currently hitting white space we are in a word, the opposite is true if at the present index we are hitting white space.

What denotes a word? In our algorithm each word is separated by one or more occurrences of white space. We don't take into account any particular splitting symbols you may use, e.g. in .NET *String.Split*[1] can take a char (or array of characters) that determines a delimiter to use to split the characters within the string into chunks of strings, resulting in an array of sub-strings.

In Figure 11.2 we present a string indexed as an array. Typically the pattern is the same for most words, delimited by a single occurrence of white space. Figure 11.3 shows the same string, with the same number of words but with varying white space splitting them.

[1] http://msdn.microsoft.com/en-us/library/system.string.split.aspx

```
1) algorithm WordCount(value)
2)     Pre:  value ≠ ∅
3)     Post: the number of words contained within value is determined
4)     inWord ← true
5)     wordCount ← 0
6)     index ← 0
7)     // skip initial white space
8)     while value[index] = whitespace and index < value.Length −1
9)         index ← index + 1
10)    end while
11)    // was the string just whitespace?
12)    if index = value.Length and value[index] = whitespace
13)        return 0
14)    end if
15)    while index < value.Length
16)        if value[index] = whitespace
17)            // skip all whitespace
18)            while value[index] = whitespace and index < value.Length −1
19)                index ← index + 1
20)            end while
21)            inWord ← false
22)            wordCount ← wordCount + 1
23)        else
24)            inWord ← true
25)        end if
26)        index ← index + 1
27)    end while
28)    // last word may have not been followed by whitespace
29)    if inWord
30)        wordCount ← wordCount + 1
31)    end if
32)    return wordCount
33) end WordCount
```

11.4 Determining the number of repeated words within a string

With the help of an unordered set, and an algorithm that can split the words within a string using a specified delimiter this algorithm is straightforward to implement. If we split all the words using a single occurrence of white space as our delimiter we get all the words within the string back as elements of an array. Then if we iterate through these words adding them to a set which contains only unique strings we can attain the number of unique words from the string. All that is left to do is subtract the unique word count from the total number of stings contained in the array returned from the split operation. The split operation that we refer to is the same as that mentioned in §11.3.

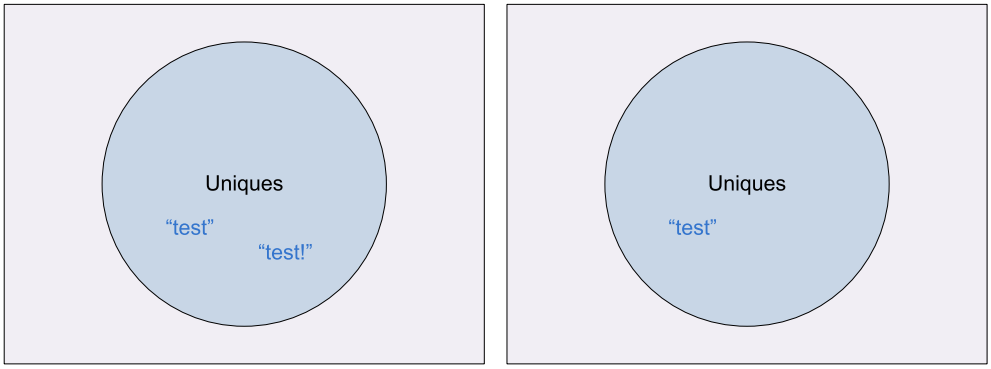

Figure 11.4: a) Undesired *uniques* set; b) desired *uniques* set

```
1) algorithm RepeatedWordCount(value)
2)     Pre:   value ≠ ∅
3)     Post:  the number of repeated words in value is returned
4)     words ← value.Split(' ')
5)     uniques ← Set
6)     foreach word in words
7)         uniques.Add(word.Strip())
8)     end foreach
9)     return words.Length −uniques.Count
10) end RepeatedWordCount
```

You will notice in the *RepeatedWordCount* algorithm that we use the *Strip* method we referred to earlier in §11.1. This simply removes any punctuation from a *word*. The reason we perform this operation on each *word* is so that we can build a more accurate unique string collection, e.g. “test”, and “test!” are the same word minus the punctuation. Figure 11.4 shows the undesired and desired sets for the *unique* set respectively.

11.5 Determining the first matching character between two strings

The algorithm to determine whether any character of a string matches any of the characters in another string is pretty trivial. Put simply, we can parse the strings considered using a double loop and check, discarding punctuation, the equality between any characters thus returning a non-negative index that represents the location of the first character in the match (Figure 11.5); otherwise we return -1 if no match occurs. This approach exhibit a run time complexity of $O(n^2)$.

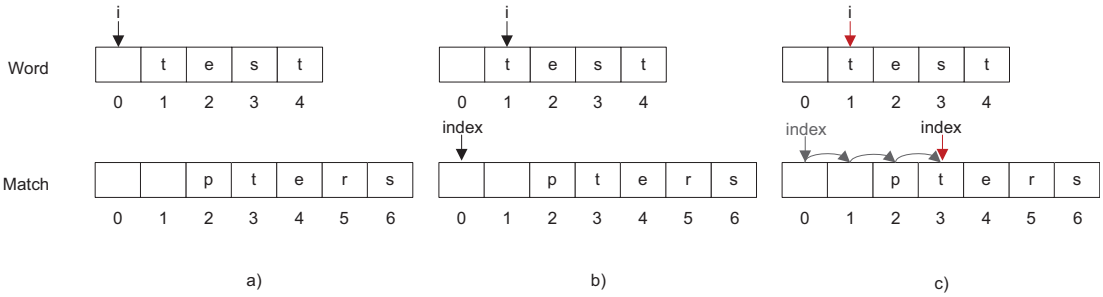

Figure 11.5: a) First Step; b) Second Step c) Match Occurred

```
1) algorithm Any(word,match)
2)     Pre:  word, match  ≠ ∅
3)     Post: index representing match location if occured, −1 otherwise
4)     for i ← 0 to word.Length − 1
5)         while word[i] = whitespace
6)                 i ← i + 1
7)         end while
8)         for index ← 0 to match.Length − 1
9)             while match[index] = whitespace
10)                    index ← index + 1
11)            end while
12)            if match[index] = word[i]
13)              return index
14)            end if
15)        end for
16)    end for
17)    return −1
18) end Any
```

11.6 Summary

We hope that the reader has seen how fun algorithms on string data types are. Strings are probably the most common data type (and data structure - remember we are dealing with an array) that you will work with so its important that you learn to be creative with them. We for one find strings fascinating. A simple Google search on string nuances between languages and encodings will provide you with a great number of problems. Now that we have spurred you along a little with our introductory algorithms you can devise some of your own.