Chapter 2
By Eric Schles

problem 2.17

2.17

Additional routines are required for the string class so that temporaries are not created when a char * is involved.

a. For the class interface presented in Figure 2.22, how many additional routines are needed?

There are 37 routines missing from the class presented in the text according to
http://www.cplusplus.com/reference/string/basic_string/

b.

I will implement size, empty, clear and at operators. (See ch2q17.cpp for implementaion)

problem 2.18

2.18 Define operator() to return a substring.
a. What is the return type?

The return type is a string.

b. implement the substring operator.
The answer is implemented in ch2q18.cpp

c. Is there a substantial difference between the following two alternatives?

```
string  subStr = s (1, 2);

//And

string subStr;
subStr = s(1,2);
```

The first calls a copy constructor the second calls the assignment operator. I guess it's not really that different.

problem 2.19

2.19 Let s be a string

a. Is the typical C mistake s= 'a' caught by the compiler? Why or why not?

No because there is an implicit type conversion from C-string to string.

b. What functions are called in s+='a'?

The += operator is called as well as the function that does the type conversion from c-string to string.

problem 2.20

2.20

Suppose that we add a constructor allowing the user to specify the initial size for the internal buffer. Describe an implementation of this construct and then explain what happens when the user attempts to declare a string with a buffer size of 0.

declaration:
string(const string & str, int bufferSize);

if the user tries to declare a buffer of size zero the function should have a condition check that raises an error. All strings must contain '\0' and thus buffer must be at least size 1.

Chapter 5

problem 5.8

see ch5q8

problem 5.9

see ch5q9

Chapter 6

problem 6.12

An algorithm takes .5 ms(milliseconds, a millisecond is a 1/1000 seconds) for input size 100. How large a problem can be solved in 1 min if the runing time is
a. Linear

1 minute = 60 seconds.

There are 1000 miliseconds in a second => 60,000 milliseconds in a minute.

Since the algorithm takes .5 ms => 120,000 problems can be solved in 1 min.

b. O(N log N)

Still 60,000 milliseconds in a minute.

Algorithm takes .5 ms, scales at a rate of n log n => which is approximately 10 times as slow.

algorithm takes 10 times as long => 120,000/10~ 12,000 problems can be solved in 1 min.

c. quadratic

Algorithm takes .5 ms, scales at a rate of n^2 => which is approximately 100 times as slow.

algorithm takes 100 times longer as input grows.

.5 * 100 => 120000/100 = 1,200 problems can be solved in 1 min.

d. cubic

Same logic as above 1000

.5 * 10,000 => 120,000/1,000 = 120 problems can be solved in 1 min.

problem 6.13

fill in the graph for O(N^3) algorithm:

n = 10,000

First we figure out what the extra running time should be:

O(10,000N^3) => (10,000^3) * N * O(N^3) => 1,000,000,000,000 * .000009 = 9,000,000

n = 100,000

This should be 100,000 ^ 3 * .000009 = 9000000000

All algorithms:

for 10,000,000

O(N^3):

10,000,000^3 * .000009 = 9000000000000000

O(N^2)

.000004 * 10,000,000 ^ 3 = 4000000000000000

O(N)

.0000003 * 10,000,000 = 3

O(N log N)

.000006 * 10,000,000 * log(10,000,000) = 360

problem 6.14

37, 2/N, N, N log log N, N log N, N log^2 N, N log N^2, N ^ ½, N ^ 1.5, N ^2, N^2 log N, N^3, 2 ^ N/2, 2^N

problem 6.15

A.
Fragment # 1 is O (N)

Fragment #2 is O(N)

Fragment #3 is O(N^2)

Fragment #4 is O(N)

Fragment #5 is O(N^3)

Fragment #6 is O(N^2)

Fragment #7 is O(N^4)

B.
see ch6q15 for code.

Output found here:

n = 10

zero seconds for all fragments

n = 100

zero seconds for all fragments except #7, #7 took 12.58000 seconds

n = 1000

zero for all except fragment #5 and #7, fragment #5 took 3.12 seconds. Fragment #7 not applicable. Ran longer than 5 minutes.

n = 10,000

fragments #5 and #7 ran longer than 5 minutes. Fragment #3 took .25 seconds, fragment #6 took .13 seconds. The other fragments took zero seconds.

n = 100,000

fragments #5 and #7 took longer than five minutes. Fragment #3 took 25 seconds, fragment #6 took 12 seconds, #1,#2 and #4 took less than zero seconds.

C.

Fragment #1

Fragment #1 runs in O(N) time. This appears consistent from the sample sizes for n. The running time did not seem to increase as the input size grew.

Fragment #2

Fragment #2 runs in O(N) time. This appears consistent from the sample sizes for n. The running time did not seem to increase as the input size grew.

Fragment #3

Fragment #3 runs in O(N^2) time. This appears to be consistent from the same sizes for n. Initially, for values of input less than 10,000 #3 appears to run faster or as fast as O(N). However as input sizes grew running time slowed down. This is consistent with the graph expected for a O(N^2) algorithm.

Fragment #4

Fragment #4 runs in O(N) time. This appears consistent from the sample sizes for n. The running time did not seem to increase as the input size grew.

Fragment #5

Fragment #5 runs in O(N^3) time. This appears to be consistent from the same sizes for n. Initially, for values of input less than 1,000 #5 appears to run faster or as fast as O(N). However as input sizes grew running time slowed down. This is consistent with the graph expected for a O(N^3) algorithm. Notice that this algorithm explodes at n = 10,000 and above.

Fragment #6

Fragment #6 runs in O(N^2) time. This appears to be consistent from the same sizes for n. Initially, for values of input less than 10,000 #6 appears to run faster or as fast as O(N). However as input sizes grew running time slowed down. This is consistent with the graph expected for a O(N^2) algorithm. Notice that this algorithm is about twice as fast as some of the other O(N^2) algorithms. This implies there is a bit of variance in the classification of these Big-Oh running times, as one would expect, given the sloppy nature of the identifier.

Fragment #7

Fragment #7 runs in O(N^4) time. This appears to be consistent from the same sizes for n. Initially, for values of input less than 100 #7 appears to run faster or as fast as O(N). However as input sizes grew running time slowed down. This is consistent with the graph expected for a O(N^4) algorithm. Notice that this algorithm explodes at n = 1,000 and above.

Problem 6.16

a.

This fragment runs in O(N^4) if you multiply each of the for – loops. However it is probably going to run in O(N^3) time.

b.

see ch6q16.cpp for code.
n = 10

fragment runs in 0 seconds.

N = 100

fragment runs in .03 seconds.

N = 1000

fragment took more than five minutes to run.

For all other N's took more than five minutes to run.

c.

Even though this algorithm is a O(N^4) algorithm it runs more like a O(N^3) algorithm since the inner most for -loop is rarely checked. This is consistent with the running times for the other O(N^3) algorithm.