Analysis:

1) The SimpleStrand algorithm has a Big Oh runtime of O(n+Bl), where n = length of the original strand, B = number of enzyme sites, and l = length of splicee strand.
This relationship can be demonstrated empirically by varying n, B, and l, and examining the resulting runtime.

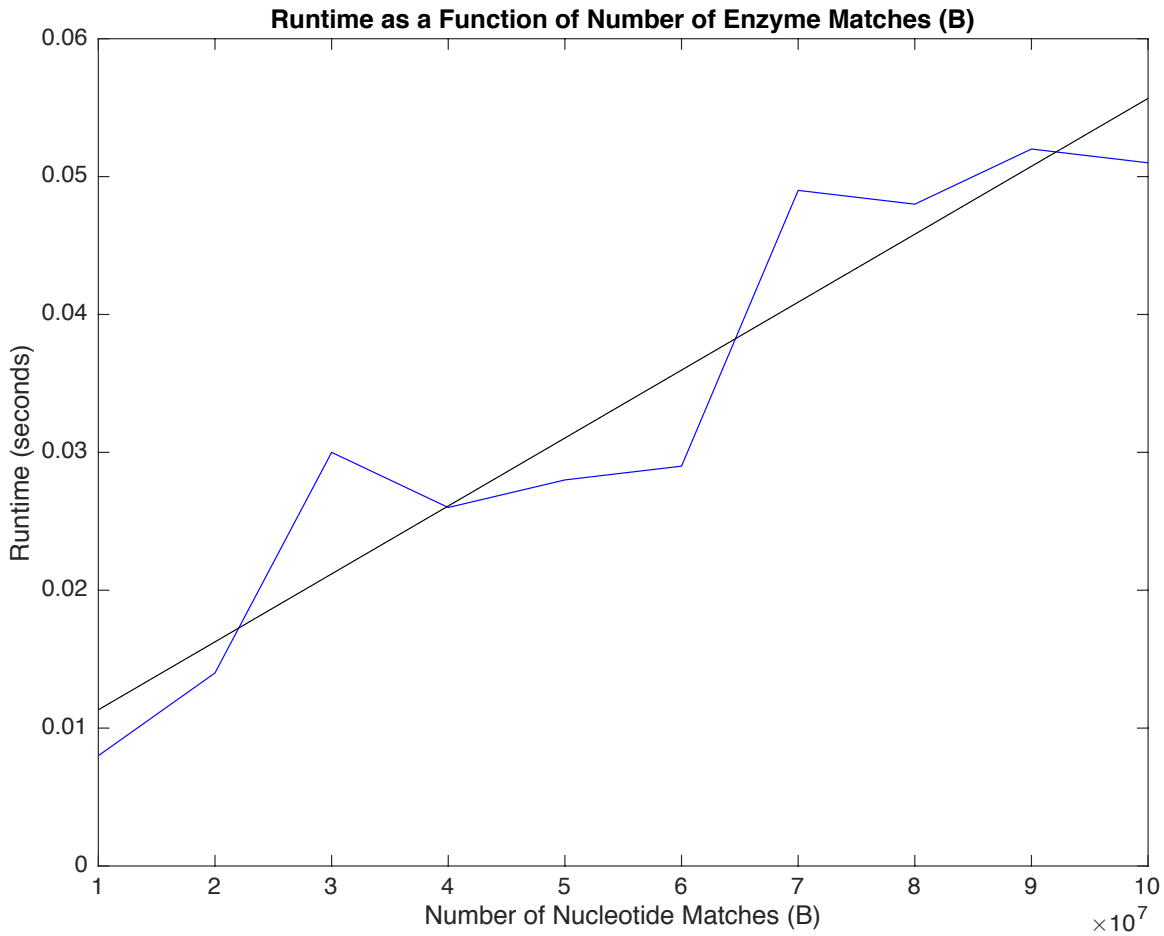
To do this, I first varied the length of the splicee (l), by running the DNABenchmark on ecoli_small.txt.
The results are shown below:

```
dna length = 320,160
cutting at enzyme gaattc
-----
Class                 splicee       recomb      time
-----
SimpleStrand:                 256           331,410   0.002 # append calls = 90
SimpleStrand:                 512           342,930   0.002 # append calls = 90
SimpleStrand:               1,024           365,970   0.002 # append calls = 90
SimpleStrand:               2,048           412,050   0.002 # append calls = 90
SimpleStrand:               4,096           504,210   0.002 # append calls = 90
SimpleStrand:               8,192           688,530   0.004 # append calls = 90
SimpleStrand:              16,384         1,057,170   0.003 # append calls = 90
SimpleStrand:              32,768         1,794,450   0.007 # append calls = 90
SimpleStrand:              65,536         3,269,010   0.007 # append calls = 90
SimpleStrand:             131,072         6,218,130   0.013 # append calls = 90
SimpleStrand:             262,144        12,116,370   0.036 # append calls = 90
SimpleStrand:             524,288        23,912,850   0.046 # append calls = 90
SimpleStrand:           1,048,576        47,505,810   0.081 # append calls = 90
SimpleStrand:           2,097,152        94,691,730   0.262 # append calls = 90
SimpleStrand:           4,194,304       189,063,570   0.284 # append calls = 90
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
      at java.util.Arrays.copyOf(Arrays.java:3332)
      at
java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.ja
va:137)
      at
java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBu
ilder.java:121)
      at
java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:421)
      at java.lang.StringBuilder.append(StringBuilder.java:136)
      at SimpleStrand.append(SimpleStrand.java:137)
```

```
        at SimpleStrand.cutAndSplice(SimpleStrand.java:65)
        at DNABenchmark.strandSpliceBenchmark(DNABenchmark.java:71)
        at DNABenchmark.main(DNABenchmark.java:122)
```

As you can see, the length of the splicee starts at 256 and doubles each time it is run. The runtime increases linearly with the length of the splicee.
While, for short splicees, the runtime does not increase when the splicee length doubles, that could be because it is simply very fast with short splicees, and the measurements are not precise enough.
For longer splicees, (starting at about l = 65,536), the runtime doubles each time the splicee length doubles, thus implying a linear relationship.

To test the relationship between runtime and number of enzyme sites (B), I had to test the program, keeping n and l the same. To do this, I wrote a program (test.java) that created text files of a fixed length (n), where the first B entries in the file match the enzyme bonding site. In this way, I could change B without changing the length of the file.
I then ran the DNABenchmark.java on several of the files I generated, and obtained the results listed below:
SimpleStrand: using splicee 256, B increasing from 10,000 to 100,000 by 10,000, n = 1,000,000:
B = 10,000 time: 0.008
B = 20,000 time: 0.014
B = 30,000 time: 0.030
B = 40,000 time: 0.026
B = 50,000 time: 0.028
B = 60,000 time: 0.029
B = 70,000 time: 0.049
B = 80,000 time: 0.048
B = 90,000 time: 0.052
B = 100,000 time: 0.051
As you can see, the runtime increases as B increases, and when B increases by an order of magnitude (from 10,000 to 100,000), the runtime increases by an order of magnitude as well (from 0.008 to 0.051).
To better examine the relationship between the runtime and the number of restriction sites, I used Matlab to produce a graph of the data and fit a line to the data. (The script I ran in Matlab is labeled "generatePlots.m" and is included in the data folder.)
The graph created is labeled "changingB.pdf" and is included in the data folder, as well as displayed below.



The equation given by Matlab to fit the data was t=4.9273e-07B+0.0064, so as you can see, it is a linear fit.

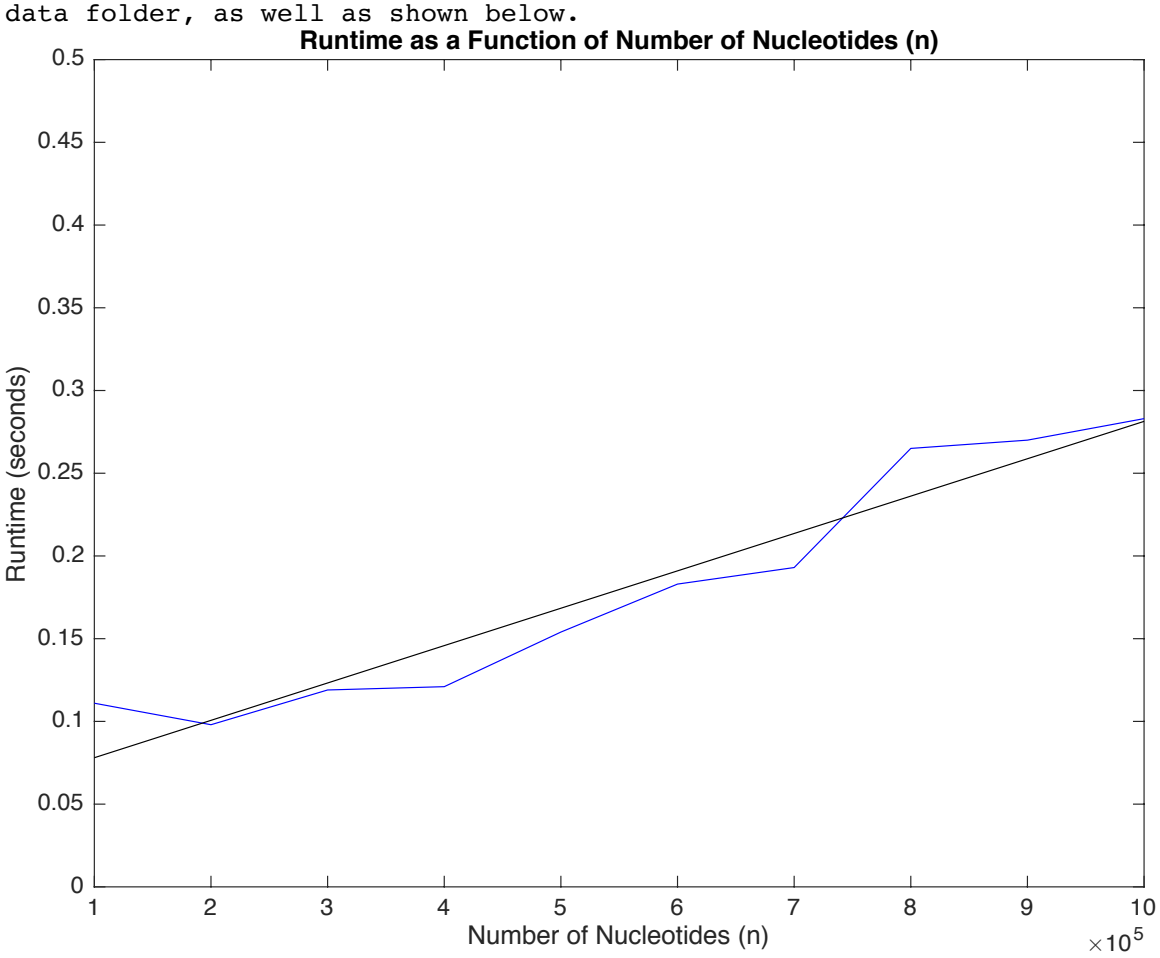
Finally, to test the relationship between runtime and length of dna file (n), I tested the program DNABenchmark, while keeping B and l the same. To do this, I used the test.java program I had written to create a list of files with constant B, but varying length n, and ran the program on them, taking the first readout (the one with l = 256). The results obtained are listed below:

SimpleStrand: using splicee 256, B = 1,000,000, n increasing from 10,000,000 to 100,000,000 by 10,000,000

n = 10,000,000 t: 0.111
n = 20,000,000 t: 0.098
n = 30,000,000 t: 0.119
n = 40,000,000 t: 0.121
n = 50,000,000 t: 0.154
n = 60,000,000 t: 0.183
n = 70,000,000 t: 0.193
n = 80,000,000 t: 0.265
n = 90,000,000 t: 0.270
n = 100,000,000 t: 0.283

As you can see, the runtime increases as n increases. To better examine the relationship between runtime and the number of nucleotides, I used Matlab to produce a graph of the data and fit a line to the data. (The script I ran in Matlab is labeled "generatePlots.m" and is included in the data folder.)

The graph created is labeled "changingn.pdf" and is included in the data folder, as well as shown below.



The equation given by Matlab is t=2.2588e-07n+0.055467, so it is evident that this is a linear fit.

2) I determined empirically that for a 512 Mb virtual machine, a splicee size of 65,536 was the largest that the computer could handle without overflowing the heap size, and that took 0.089 seconds to process.
The largest splicee size for the 1024 Mb virtual machine is 131,072, which took 0.142 seconds. The full results of the tests are printed below.
I determined that these are the largest sizes the computer could handle because the DNABenchmark class runs tests by doubling the splicee size and running SimpleStrand. Eventually, this generates an error that the computer ran out of heap space. The last method that finished before the heap space ran out, therefore, must be the largest that the computer can handle.
For each doubling of the virtual machine size, the computer could double the size of the splicee sequence that it could handle, which

makes sense, as the memory required would double when the length of the splicee doubles.
Furthermore, it makes sense that the times approximately doubled when the splicee size doubled, as doubling the splicee size approximately doubles the amount of information to be processed.
512 MB virtual machine: SimpleStrand:
65,536 46,906,071 0.089 # append calls = 1290
1024 MB virtual machine: SimpleStrand:
131,072 89,176,791 0.142 # append calls = 1290
8192 MB virtual machine: SimpleStrand:
1,048,576 680,966,871 1.352 # append calls = 1290
No modifications virtual machine: SimpleStrand: 262,144
173,718,231 0.248 # append calls = 1290

3) To determine the Big-Oh runtime of the LinkedStrand.java class, I empirically tested the class using DNABenchmark.java and changing B, n, and l and recording the resulting runtime.

The first thing I tested was the runtime when varying the length of the splicee (l).
To do this, I simply ran DNABenchmark.java with the ecoli.txt file.
The results are listed below:
LinkStrand: ecoli.txt
dna length = 4,639,221
cutting at enzyme gaattc

Class	splicee	recomb	time
LinkStrand:	256	4,800,471	0.041 # append calls = 1290
LinkStrand:	512	4,965,591	0.033 # append calls = 1290
LinkStrand:	1,024	5,295,831	0.036 # append calls = 1290
LinkStrand:	2,048	5,956,311	0.030 # append calls = 1290
LinkStrand:	4,096	7,277,271	0.029 # append calls = 1290
LinkStrand:	8,192	9,919,191	0.037 # append calls = 1290
LinkStrand:	16,384	15,203,031	0.030 # append calls = 1290
LinkStrand:	32,768	25,770,711	0.030 # append calls = 1290
LinkStrand:	65,536	46,906,071	0.030 # append calls = 1290
LinkStrand:	131,072	89,176,791	0.030 # append calls = 1290
LinkStrand:	262,144	173,718,231	0.030 # append calls = 1290
LinkStrand:	524,288	342,801,111	0.030 # append calls = 1290
LinkStrand:	1,048,576	680,966,871	0.030 # append calls = 1290
LinkStrand:	2,097,152	1,357,298,391	0.030 # append calls = 1290

```
LinkStrand:       4,194,304     2,709,961,431   0.031 # append calls =
1290
LinkStrand:       8,388,608     5,415,287,511   0.028 # append calls =
1290
LinkStrand:      16,777,216    10,825,939,671   0.028 # append calls =
1290
LinkStrand:      33,554,432    21,647,243,991   0.028 # append calls =
1290
LinkStrand:      67,108,864    43,289,852,631   0.042 # append calls =
1290
LinkStrand:     134,217,728    86,575,069,911   0.054 # append calls =
1290
LinkStrand:     268,435,456   173,145,504,471   0.040 # append calls =
1290
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
      at java.util.Arrays.copyOf(Arrays.java:3332)
      at
java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.ja
va:137)
      at
java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBu
ilder.java:121)
      at
java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:421)
      at java.lang.StringBuilder.append(StringBuilder.java:136)
      at DNABenchmark.main(DNABenchmark.java:119)
```

As you can see, the runtime is constant as the splicee length increases, because the runtime starts at 0.041 seconds with a 256 nucleotide splicee, and is 0.040 with a 268,435,456 nucleotide splicee. Therefore, the runtime with respect to the splicee is O(1), which is better than the O(n) that SimpleStrand had. One possible reason for this is that the the LinkedStrand does not have to manipulate the entire string of the existing DNA and the splicee DNA. It simply has to change the pointers on those objects so they point at each other in a list, and changing pointers is a constant time operation. Furthermore, LinkedStrand was able to use a longer splicee, as the longest strand that SimpleStrand could do was 262,144, and the LinkedStrand was able to do up to 268,435,456 before running out of heap space. One reason for this is that the program does not have to copy the entire string of the DNA every time it changes the DNA. Instead, it simply has to change one of the nodes and then change the pointers pointing to that node.

Next, I tested the runtime as a function of the number of nucleotides of the DNA (n). To do this, I followed the same procedure as for the SimpleStrand. I used a class I wrote to create text files with different n values and ran DNABenchmark on them, with LinkStrand. The following are results from DNABenchmark with LinkStrand, using l = 256, B = 100,000, n increasing from 10,000,000 to 100,000,000 by 10,000,000

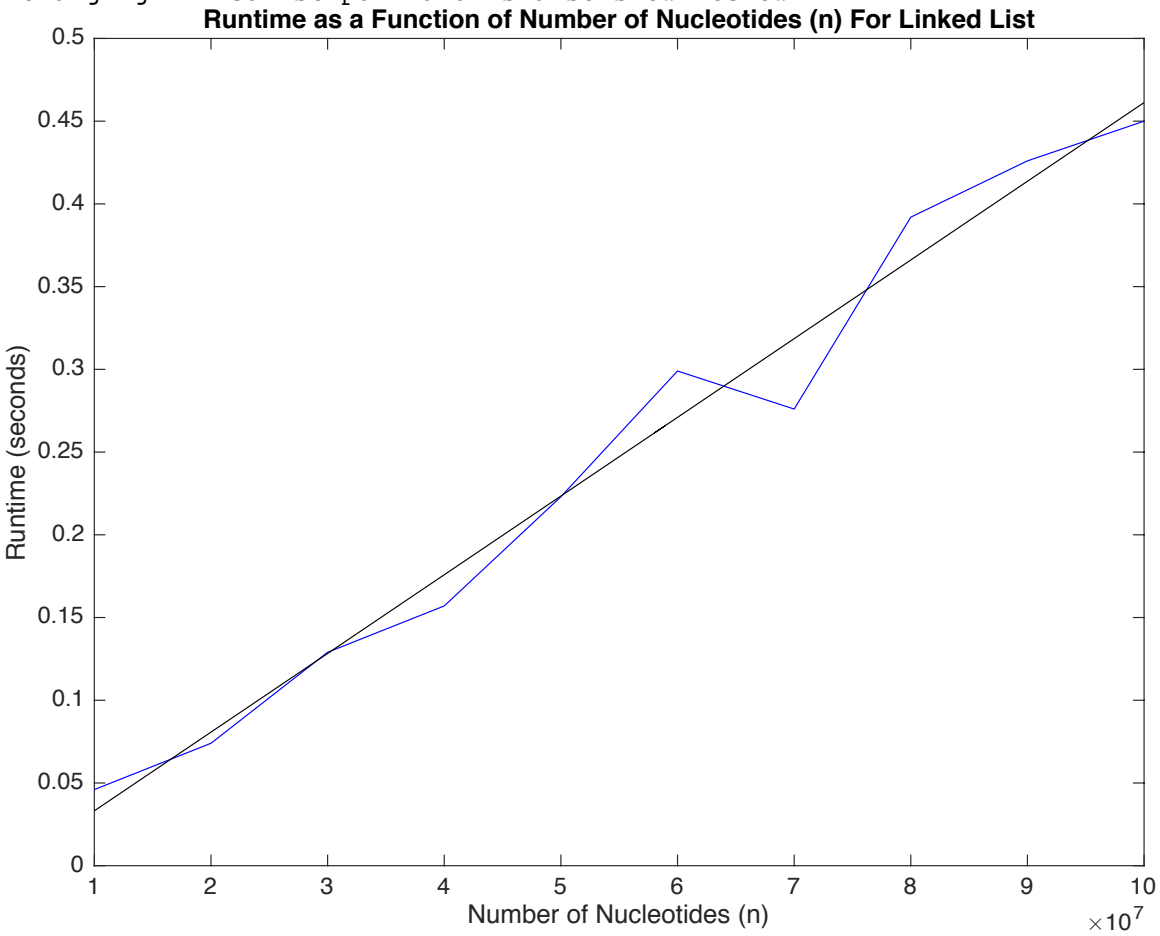
```
n = 10,000,000 t:  0.046
n = 20,000,000 t:  0.074
n = 30,000,000 t:  0.129
n = 40,000,000 t:  0.157
n = 50,000,000 t:  0.223
n = 60,000,000 t:  0.299
```

n = 70,000,000 t: 0.276
n = 80,000,000 t: 0.392
n = 90,000,000 t: 0.426
n = 100,000,000 t: 0.450

Based on these results, the Big-Oh time of the algorithm, with respect to n is linear: O(n). The time starts at 0.046 and increases by an order of magnitude (to 0.450) when n increases by an order of magnitude (from 10,000,000 to 100,000,000).

The reason that this is still O(n) is that, even though the information is stored differently, the algorithm still has to check each letter of the DNA to see if it matches the enzyme's pattern.

A graph of these results is provided in the data folder, under the name "changingnLinkedList.pdf" and is also shown below.



The regression line generated by Matlab is t=4.7564e-09n+-0.0144, so it is clearly a linear relationship.

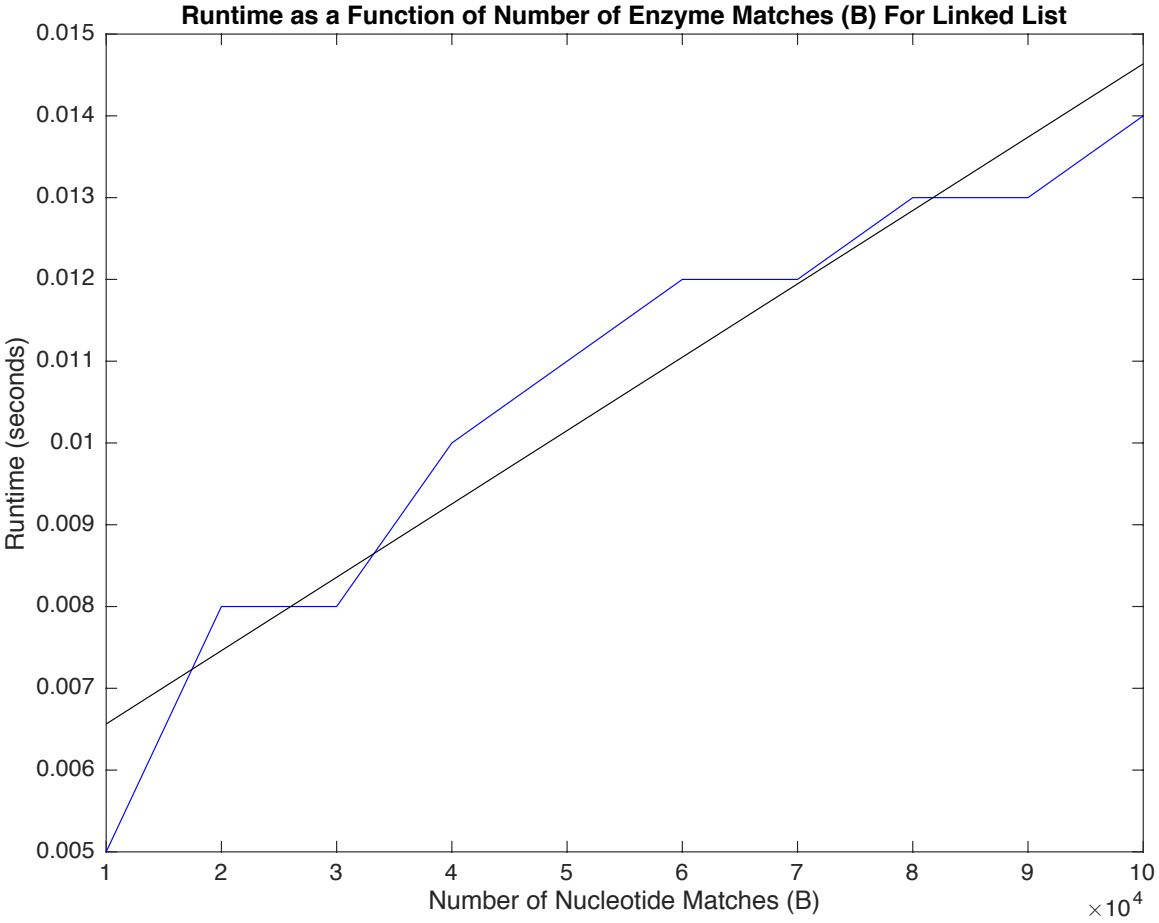
Finally, I tested the runtime as a function of the number of enzyme bonding sites in the DNA (B). To do this, I used the same procedure as above.

The results are shown below, with l = 256, n = 1,000,000, B increases from 10,000 to 100,000 by 10,000.

B = 10,000 t: 0.005
B = 20,000 t: 0.008
B = 30,000 t: 0.008
B = 40,000 t: 0.010

B = 50,000 t: 0.011
B = 60,000 t: 0.012
B = 70,000 t: 0.012
B = 80,000 t: 0.013
B = 90,000 t: 0.013
B = 100,000 t: 0.014

Based on these results, the Big-Oh runtime of the algorithm, with respect to B is linear: O(B). The reason for this is that each time the algorithm recognizes an enzyme site, it must run code to splice that site, so the more times the enzyme site is recognized, the more times it must run that code. A graph for this, titled "changingBLinkedList.pdf" is in the data folder, and is also shown below.



The regression line generated by Matlab is t=8.9697e-08B+0.0056667, so it is a linear relationship.

In summary, the LinkStrand algorithm is still affected by the length of the DNA and the number of sites, but is not affected by the length of the splicee, so the overall runtime is O(B+N).

*Note: the text files I used to get the data are not in the data folder, because they were very large text files, and I did not see a reason to upload these large files to Ambient, when the files can simply be recreated by running the test.java class.

Appendix:

Code for generatePlots.m

```
clear;
format short e;

%%changing n
n = 1e5:1e5:1e6;
time =
[0.111,0.098,0.119,0.121,0.154,0.183,0.193,0.265,0.270,0.283];%[0.009,0
.009,0.009,0.008,0.009,0.009,0.009,0.008,0.008,0.008];
figure(1);
clf;
plot(n,time,'b');
axis([-inf,inf,0,0.5]);
title('Runtime as a Function of Number of Nucleotides (n)');
xlabel('Number of Nucleotides (n)');
ylabel('Runtime (seconds)');
hold on;
p = polyfit(n,time,1);
xhat = min(n):100000:max(n);
yhat = polyval(p,xhat);
plot(xhat,yhat,'k');
hold off;
print -depsc changingn;
disp('Changing n: ');
A=['t=',num2str(p(1)),'n+',num2str(p(2))];
disp(A);

B = 1e7:1e7:1e8;
time = [0.008,0.014,0.03,0.026,0.028,0.0290,0.049,0.048,0.052,0.051];
figure(2);
clf;
plot(B,time,'b');
title('Runtime as a Function of Number of Enzyme Matches (B)');
xlabel('Number of Nucleotide Matches (B)');
ylabel('Runtime (seconds)');
hold on;
p = polyfit(B,time,1);
xhat = min(B):10000:max(B);
yhat = polyval(p,xhat);
plot(xhat,yhat,'k');
hold off;
print -depsc changingB;
disp('Changing B: ');
A=['t=',num2str(p(1)),'B+',num2str(p(2))];
disp(A);
```

Code for generatePlotsLinkedList.m

```
clear;
format short e;

%%changing n
n = 1e7:1e7:1e8;
time =
[0.046,0.074,0.129,0.157,0.223,0.299,0.276,0.392,0.426,0.450];%[0.009,0
.009,0.009,0.008,0.009,0.009,0.009,0.008,0.008,0.008];
figure(1);
clf;
plot(n,time,'b');
axis([-inf,inf,0,0.5]);
title('Runtime as a Function of Number of Nucleotides (n) For Linked
List');
xlabel('Number of Nucleotides (n)');
ylabel('Runtime (seconds)');
hold on;
p = polyfit(n,time,1);
xhat = min(n):1000000:max(n);
yhat = polyval(p,xhat);
plot(xhat,yhat,'k');
hold off;
print -depsc changingnLinkedList;
disp('Changing n: ');
A=['t=',num2str(p(1)),'n+',num2str(p(2))];
disp(A);

B = 1e4:1e4:1e5;
time = [0.005,0.008,0.008,0.010,0.011,0.012,0.012,0.013,0.013,0.014];
figure(2);
clf;
plot(B,time,'b');
title('Runtime as a Function of Number of Enzyme Matches (B) For Linked
List');
xlabel('Number of Nucleotide Matches (B)');
ylabel('Runtime (seconds)');
hold on;
p = polyfit(B,time,1);
xhat = min(B):10000:max(B);
yhat = polyval(p,xhat);
plot(xhat,yhat,'k');
hold off;
print -depsc changingBLinkedList;
disp('Changing B: ');
A=['t=',num2str(p(1)),'B+',num2str(p(2))];
disp(A);
```