SF-Seq

Part 1 - SF-Seq read quality score distributions

1.

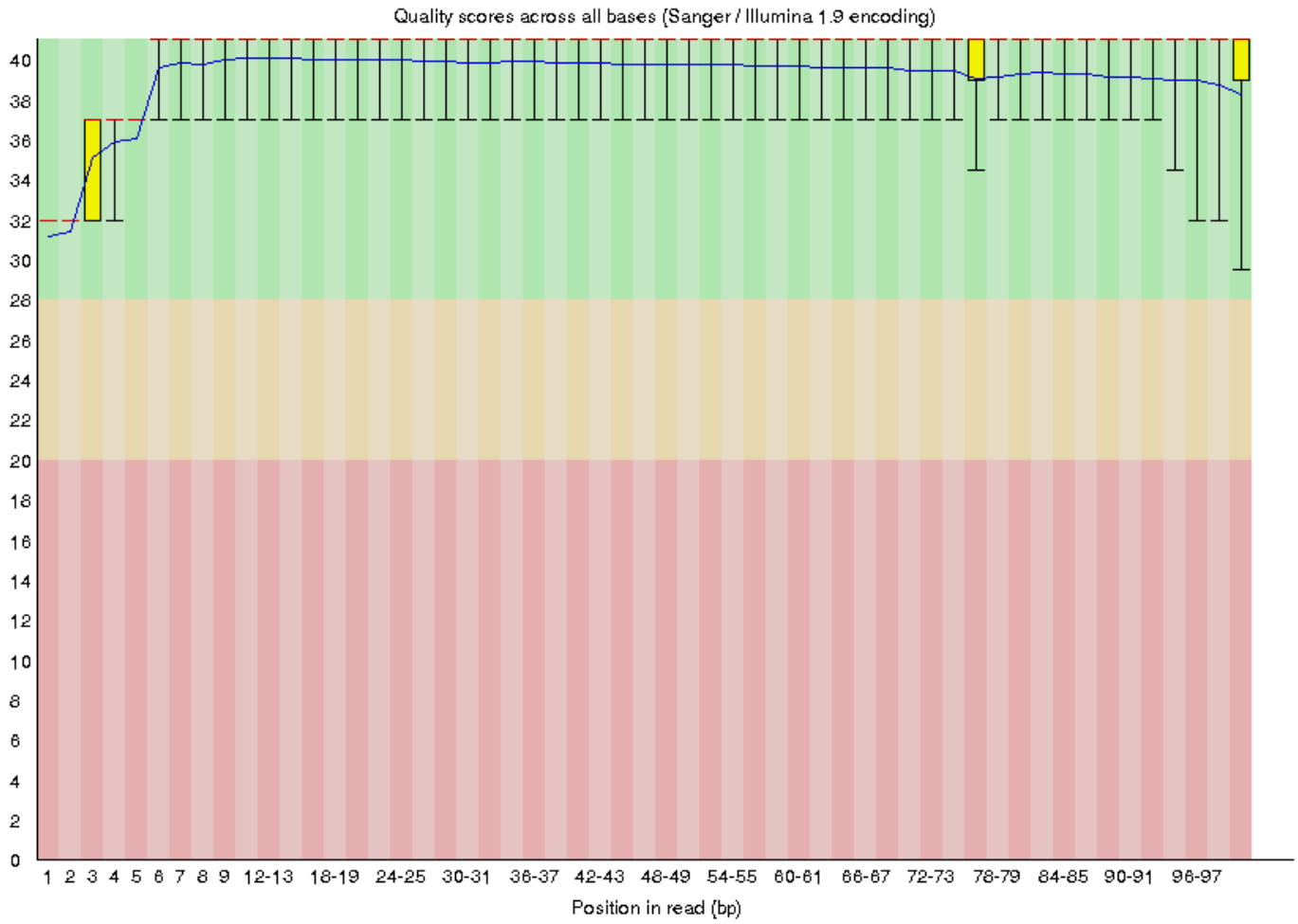
I began the process by using FastQC/0.11.5-Java-1.8.0_131 on my two libraries for both forward and reverse reads, specifying no extact so the output files would remain zipped.

Commands run for FastQC: (Commands were run on an interactive node on Talapas)

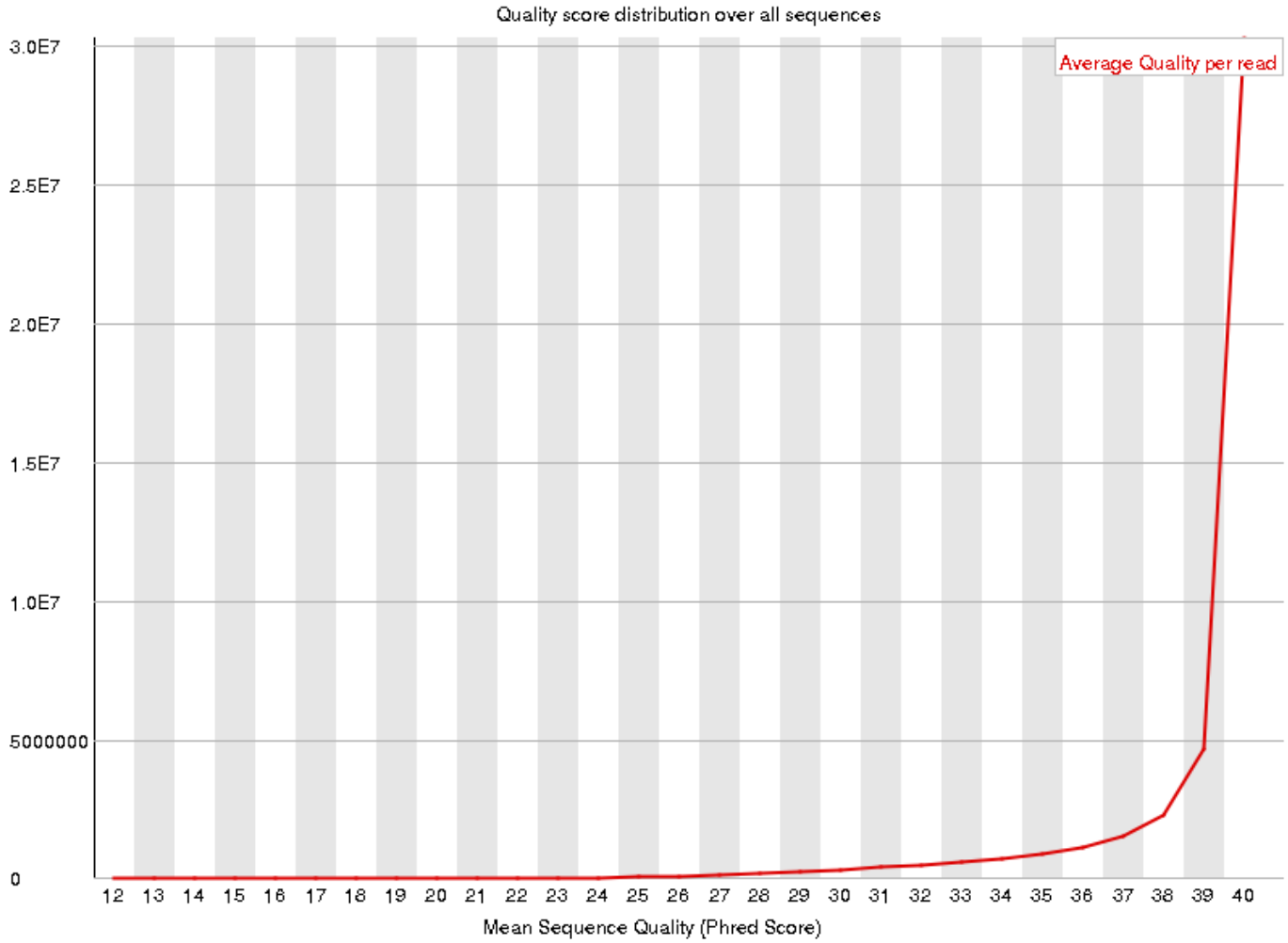
```
ml easybuild intel/2017a FastQC

fastqc -o /home/rmeng/Bi624/SF-Seq --noextract -f fastq 23_4A_control_S17_L008_R1_0
01.fastq.gz 23_4A_control_S17_L008_R2_001.fastq.gz

fastqc -o /home/rmeng/Bi624/SF-Seq --noextract -f fastq 3_2B_control_S3_L008_R1_001
.fastq.gz 3_2B_control_S3_L008_R2_001.fastq.gz
```

FastQC Graphs Generated for 23_4A_control (R1)

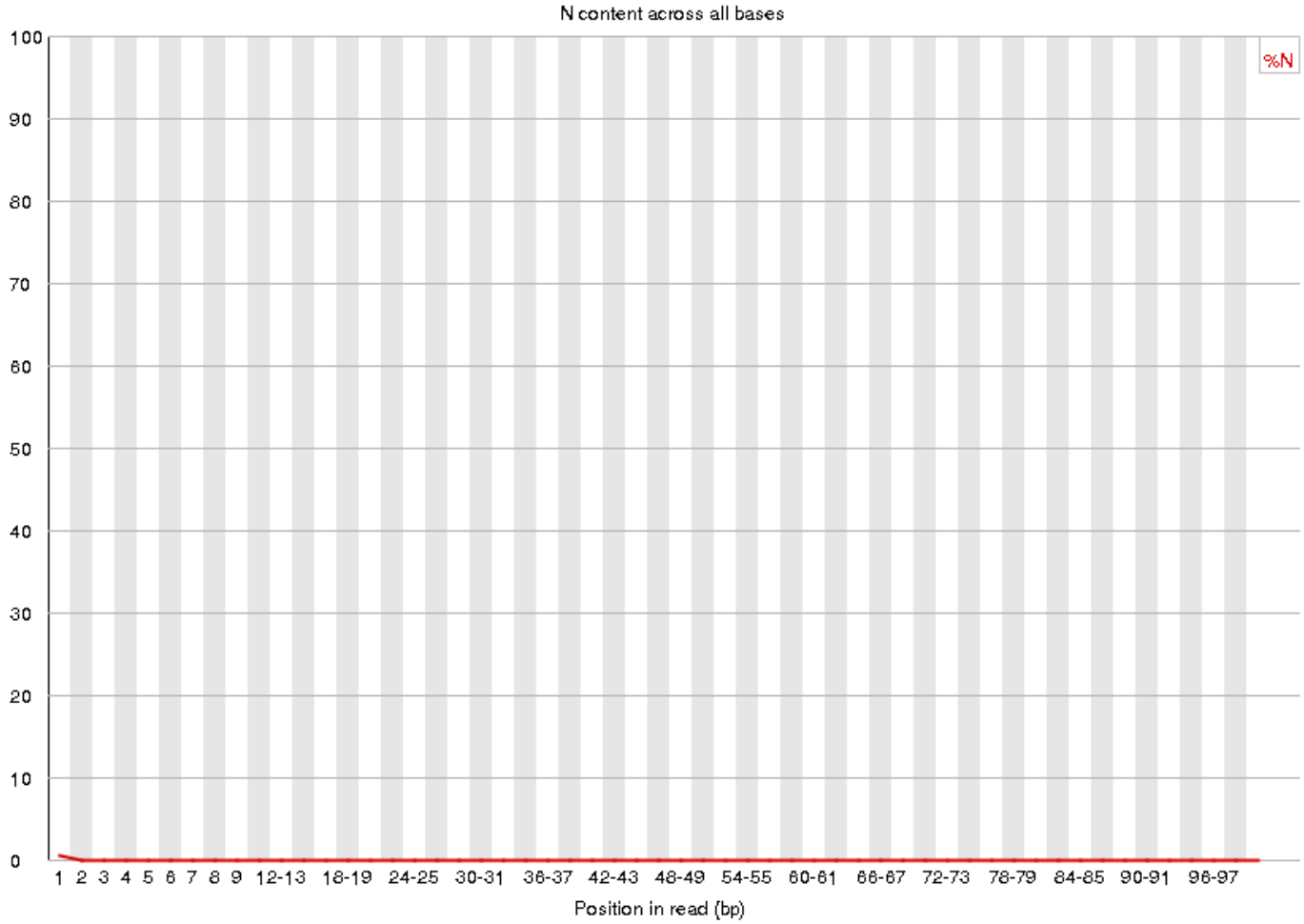
Per Base Quality for 23_4A_control (R1)



Per Sequence Quality for 23_4A_control (R1)

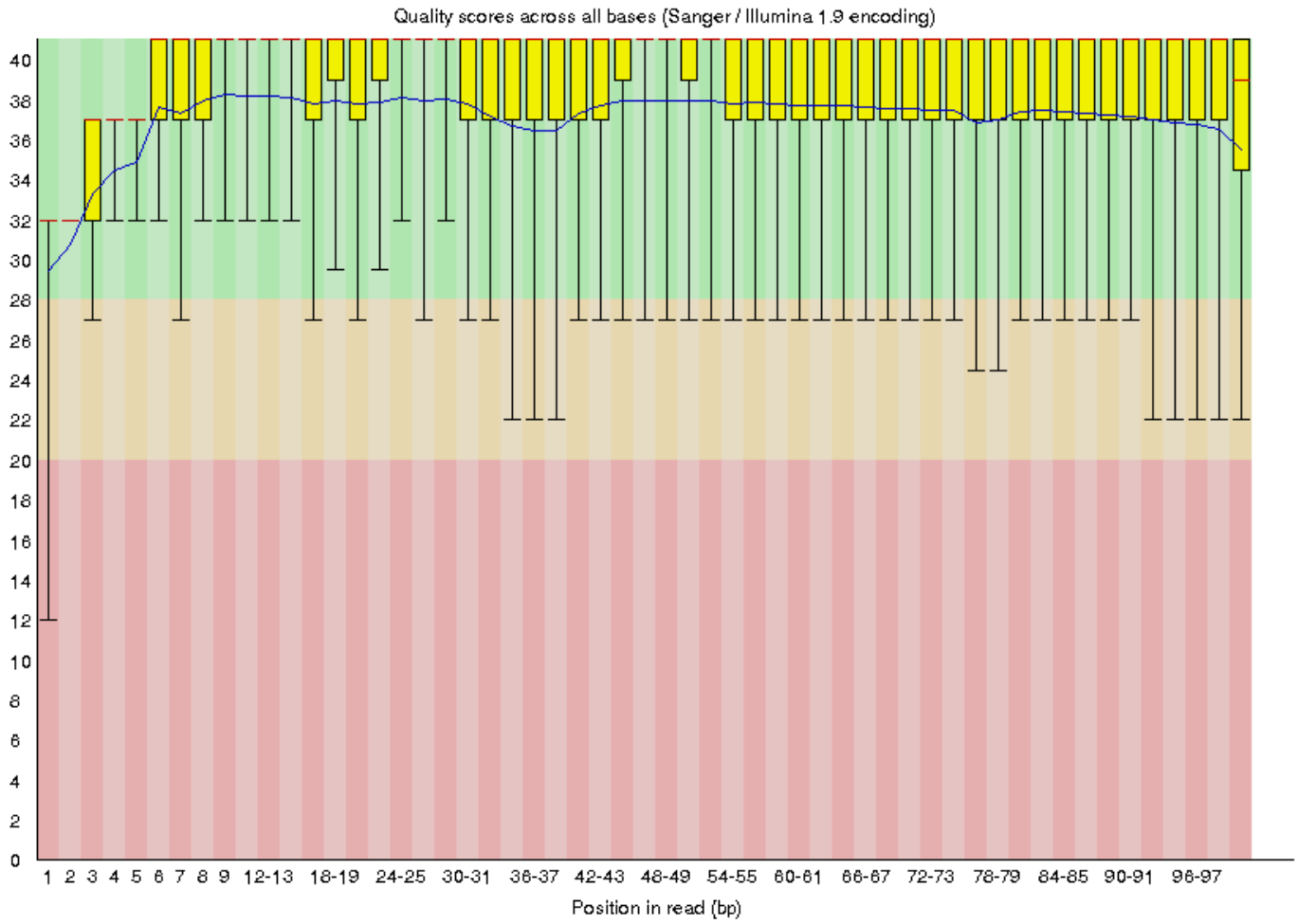


Per Base N Content for 23_4A_control (R1)

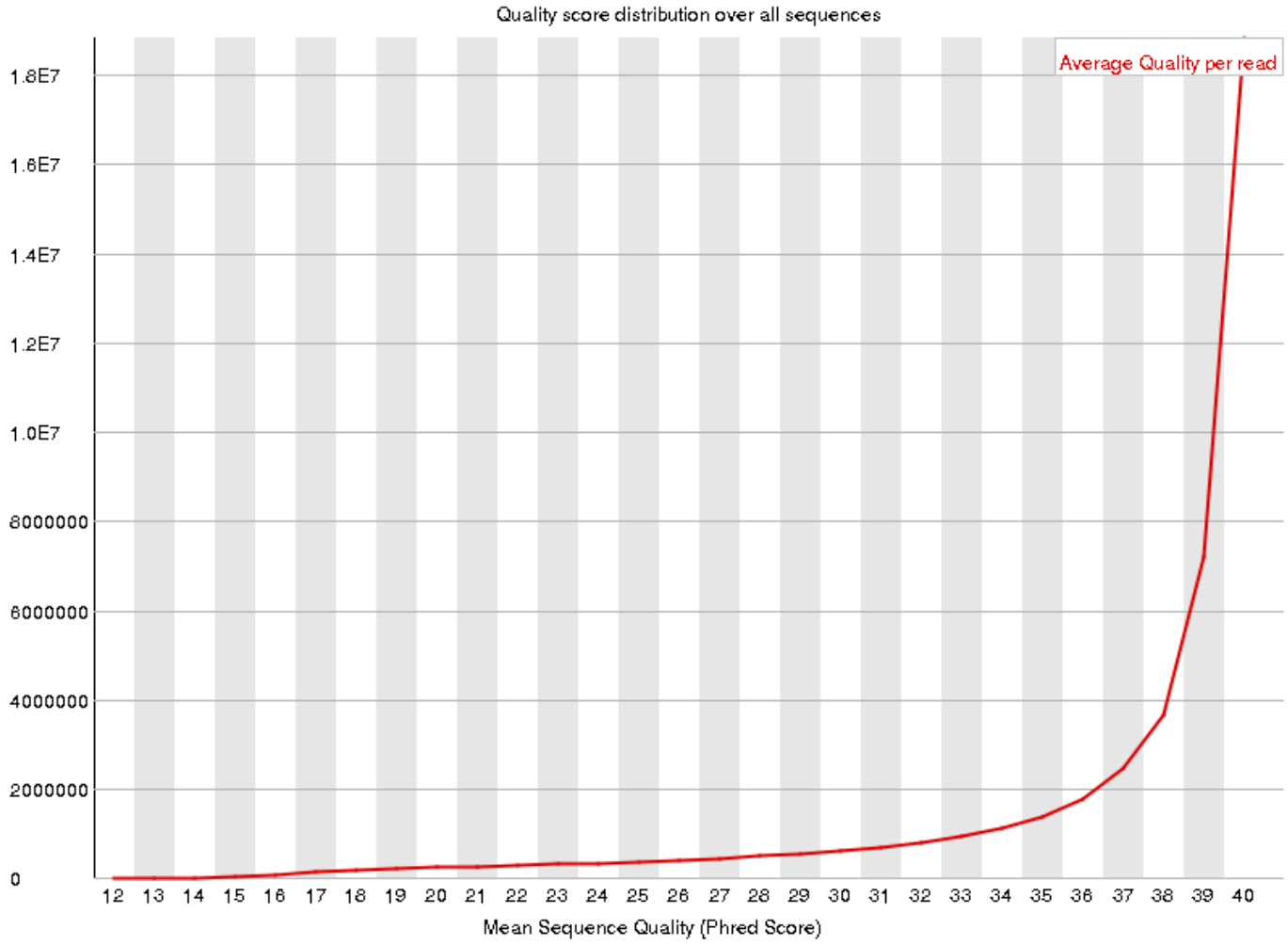


FastQC Graphs Generated for 23_4A_control (R2)

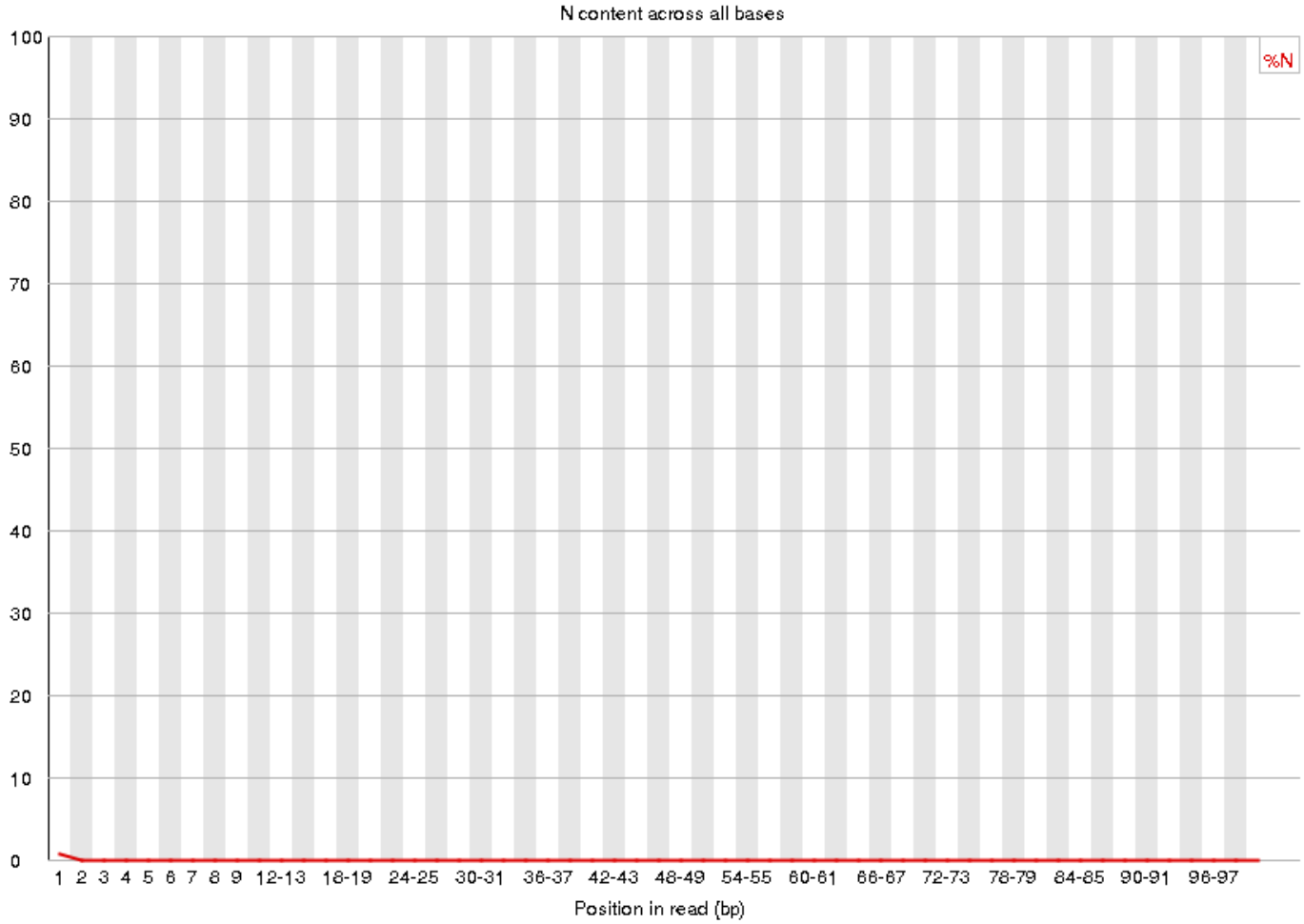
Per Base Quality for 23_4A_control (R2)



Per Sequence Quality for 23_4A_control (R2)

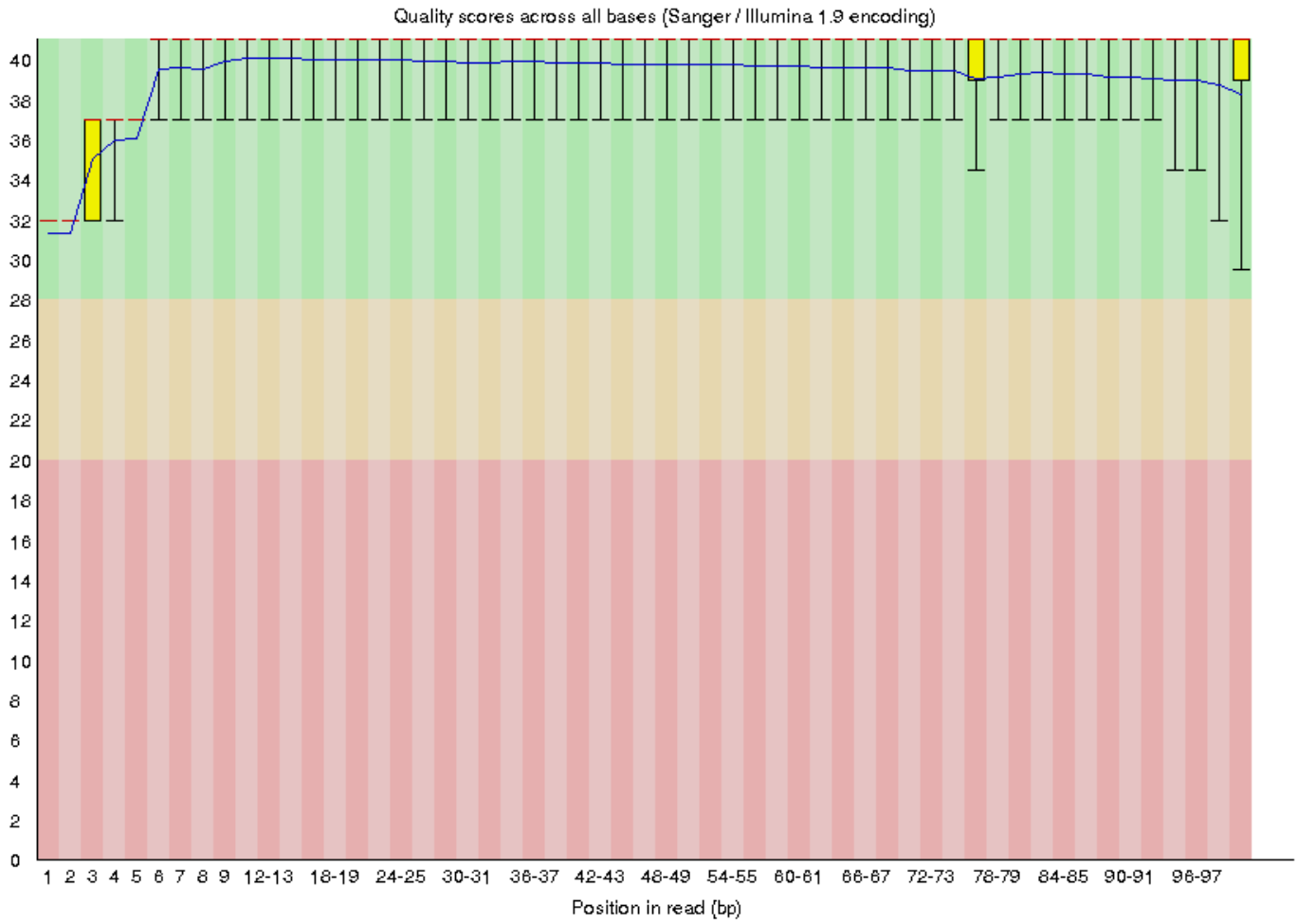


Per Base N Content for 23_4A_control (R2)

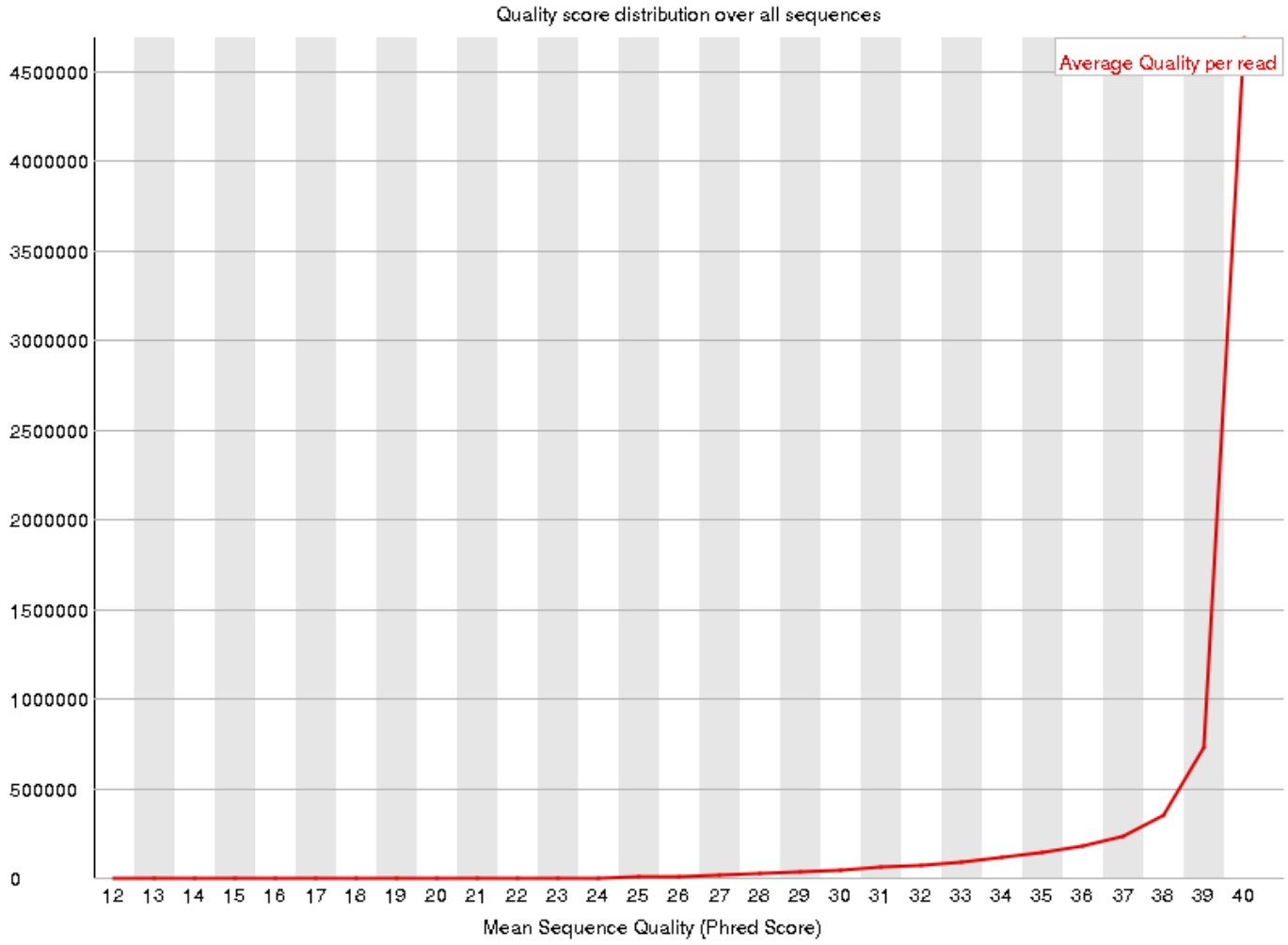


FastQC Graphs Generated for 3_2B_Control (R1)

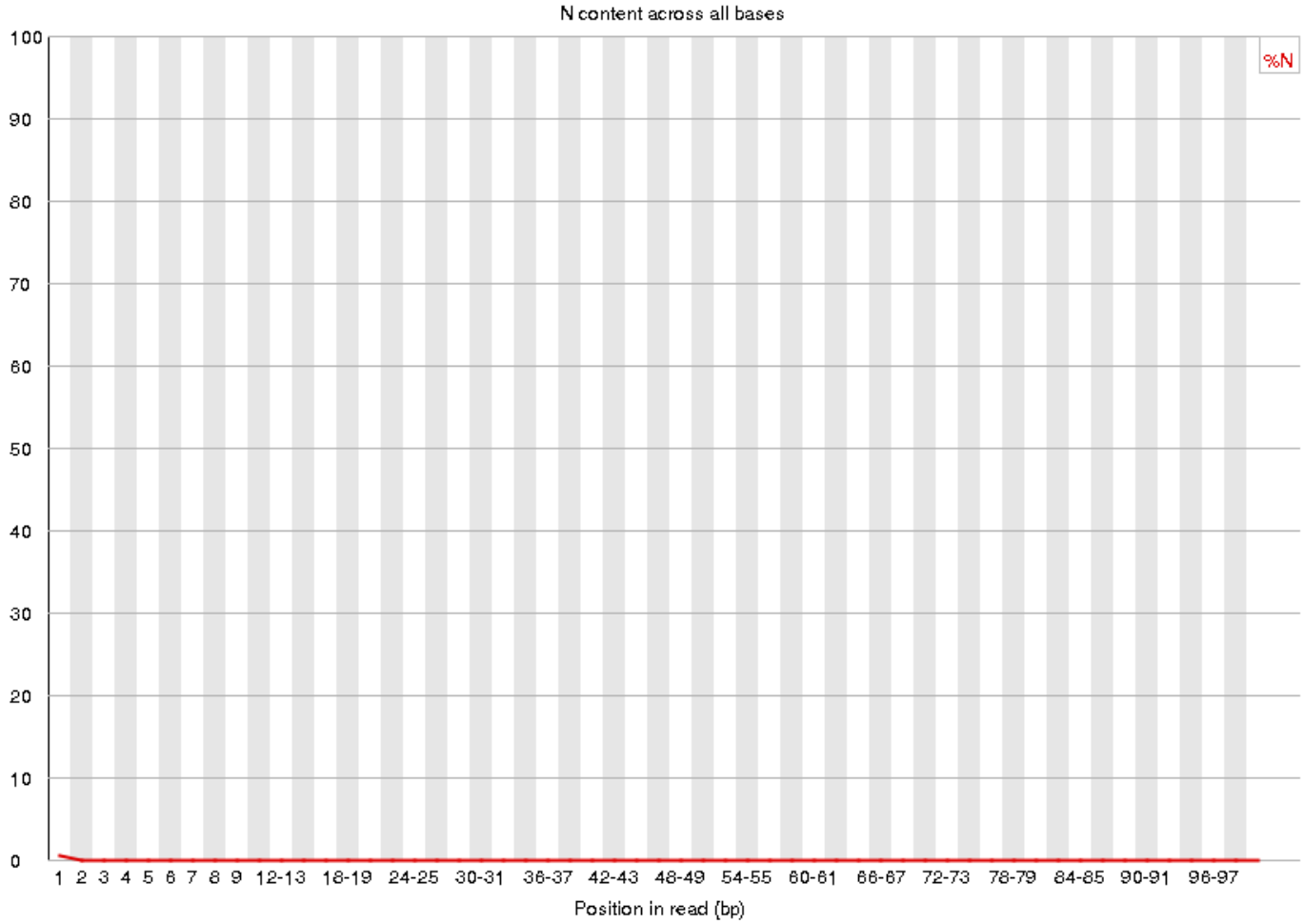
Per Base Quality for 3_2B Control (R1)



Per Sequence Quality for 3_2B Control (R1)

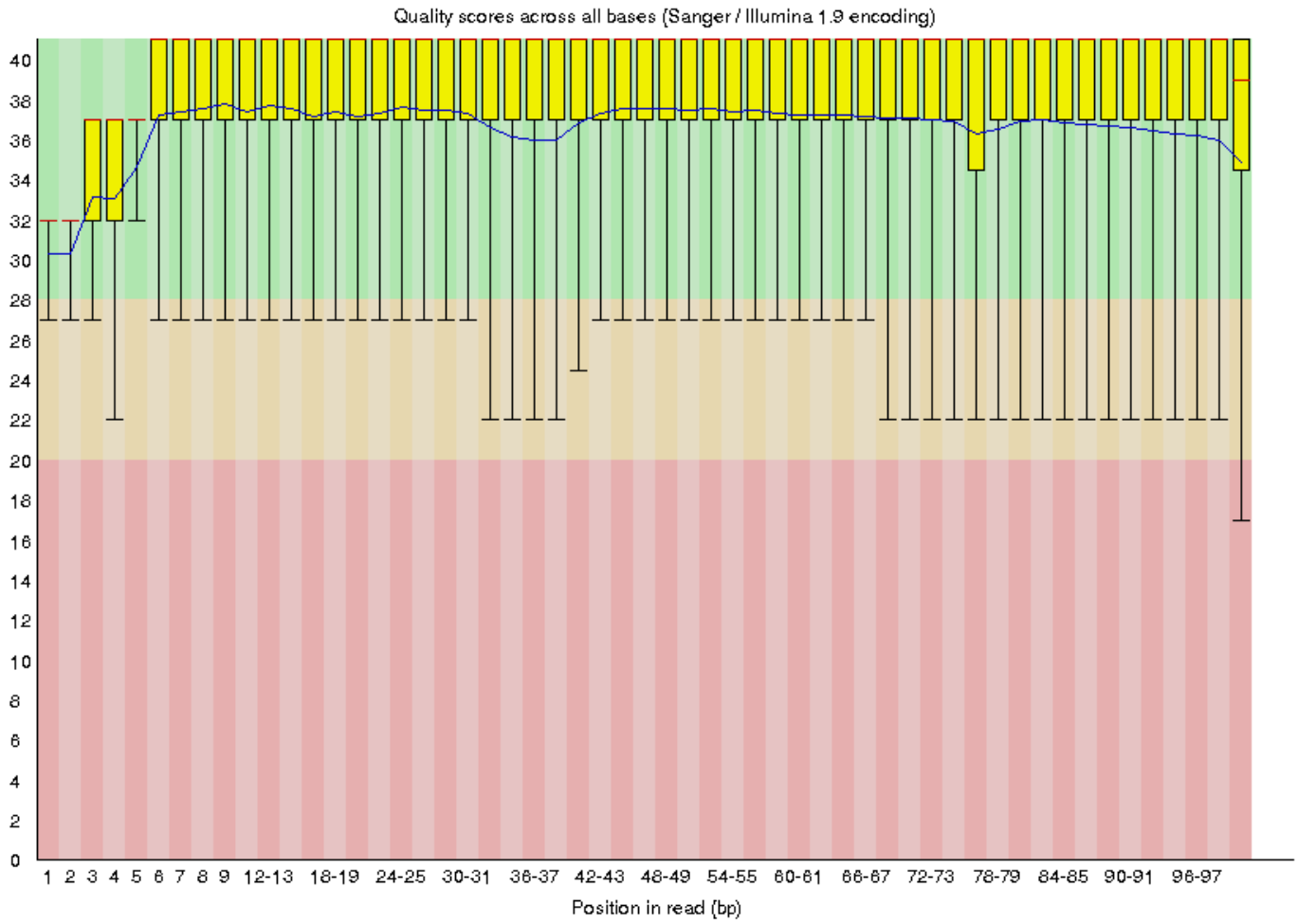


Per Base N Content for 3_2B Control (R1)

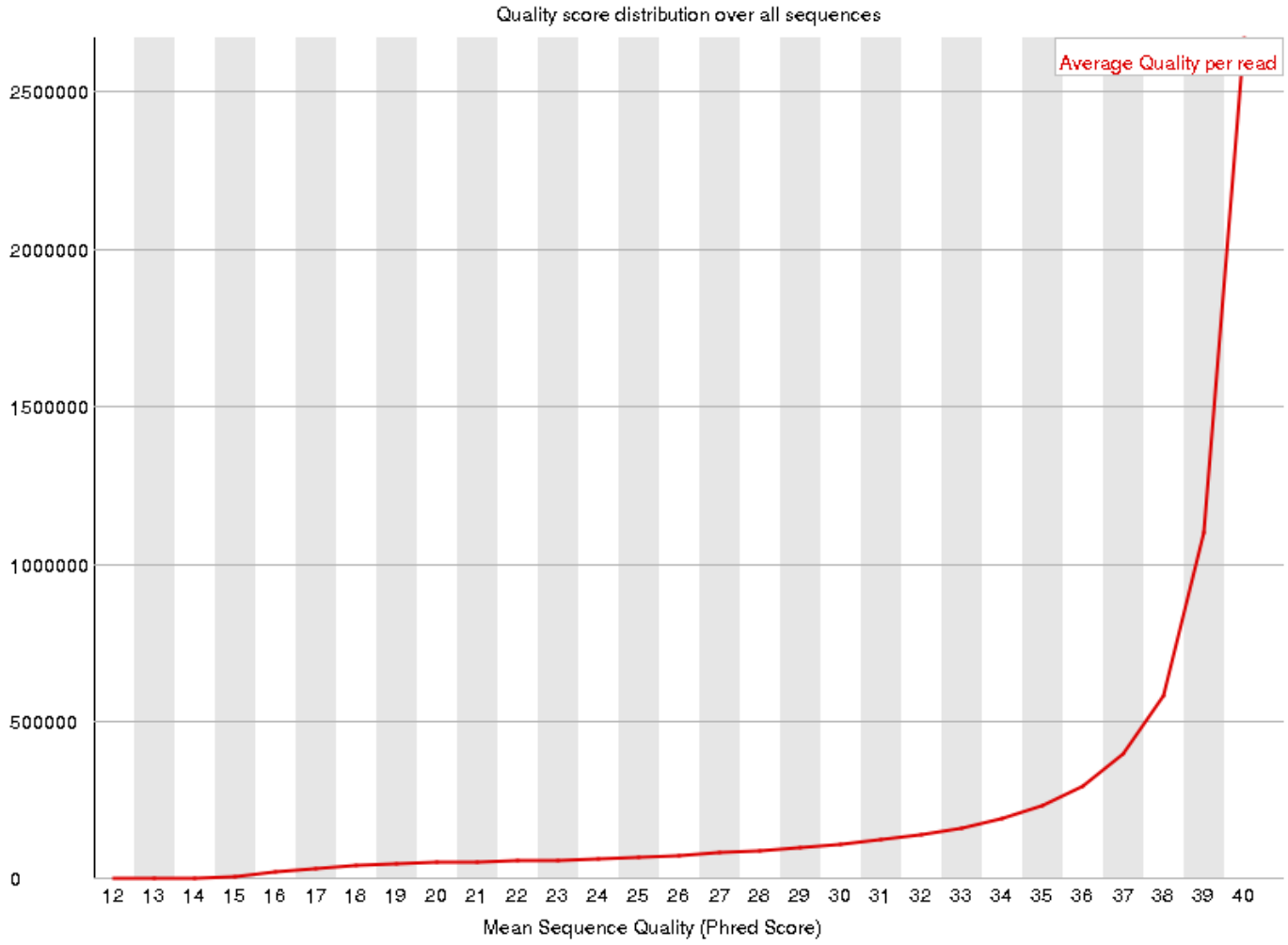


FastQC Graphs Generated for 3_2B_Control (R2)

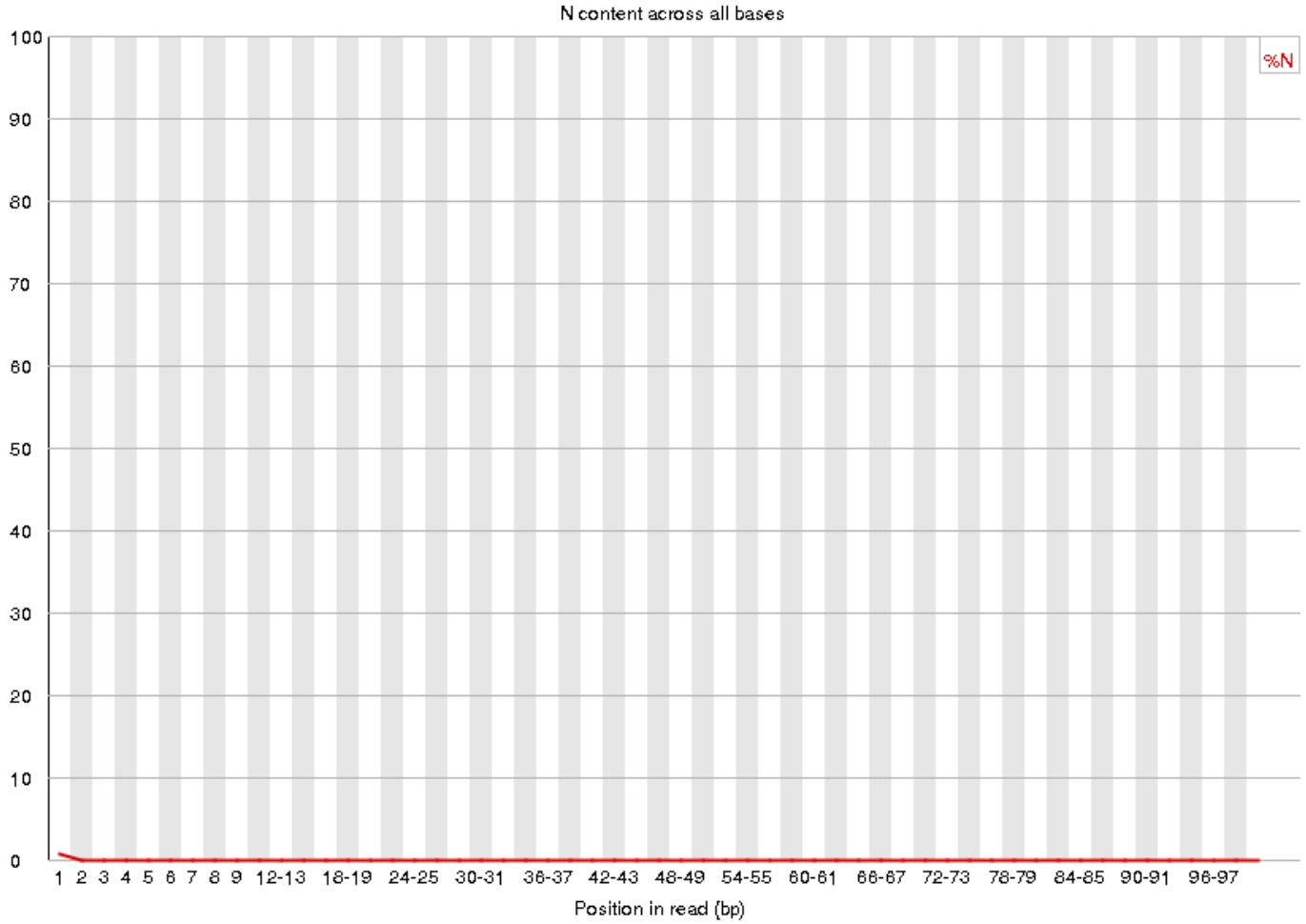
Per Base Quality for 3_2B Control (R2)



Per Sequence Quality for 3_2B Control (R2)



Per Base N Content for 3_2B Control (R2)



For both pairs of forward and reverse reads, the plots of per base N content were consistent with the quality score plots. The quality score plots showed the majority of the scores were very high quality and in turn the number of bases that appeared as an N was very low. The only basepair position that showed an N content percentage above 0 was basepair position 1 which consistently had the lowest average quality score per position in all 4 plots generated.

2.

Sample script used to generate average quality score per base per position:

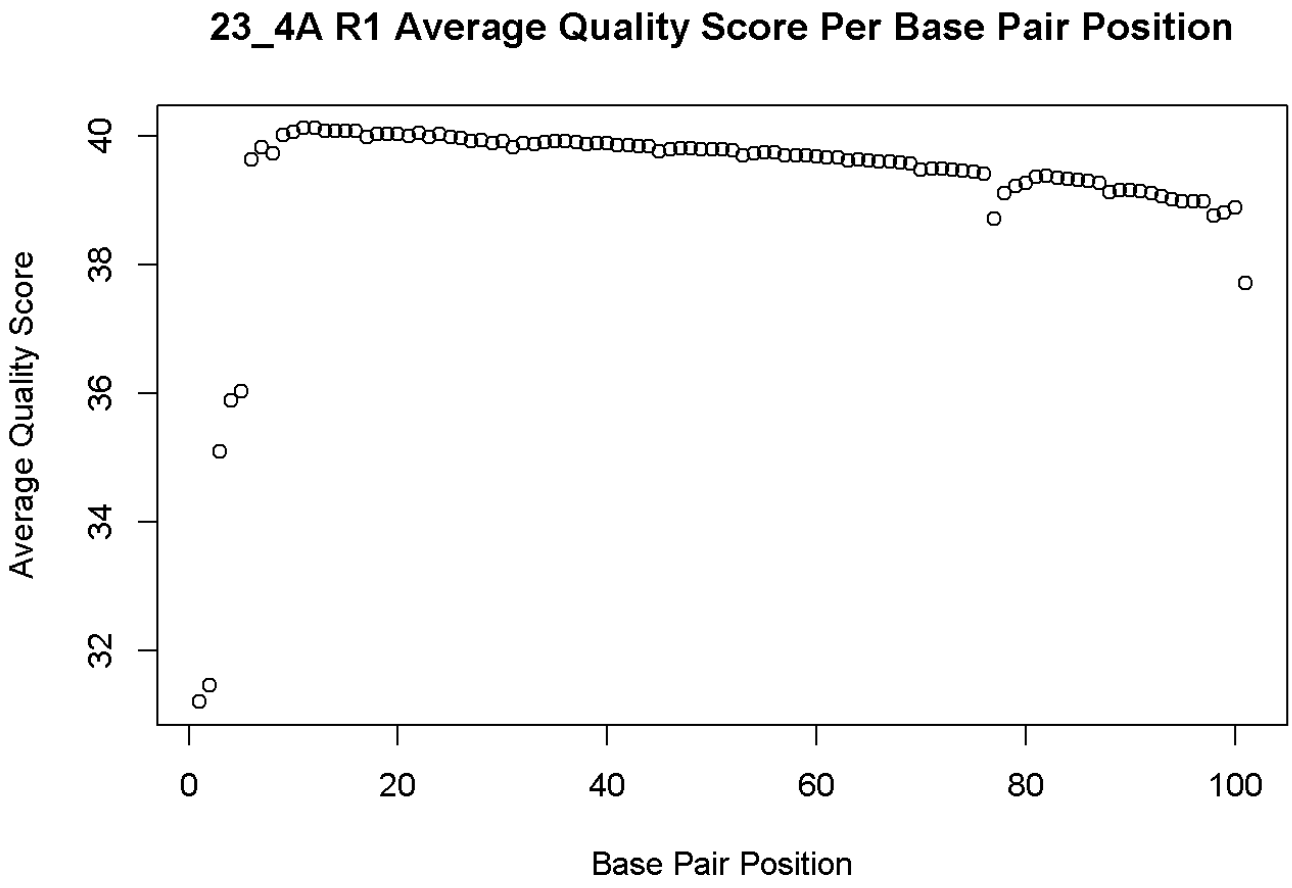
```
mean_scores =[0.0] * 101 #Generating a list of 101 values of 0.0
NR = 0 #Setting a line counter
with open('/home/rmeng/Bi624/SF-Seq/23_4A_control_S17_L008_R1_001.fastq') as fh:
    for line in fh: #Looping through the file only looking at every fourth line
        NR += 1
        line = line.strip()
        if NR % 4 == 0:
            i = 0
            for x in line: #Looping through each character in the desired line
                score = (ord(x)-33) #Calculating the quality score for said charact
er
                mean_scores[i] += score #Adding that to our list
                i += 1
    j = 0
    for x in mean_scores: #After that is done for every line, we divide by the numb
er of lines to get average.
        mean_scores[j] = mean_scores[j] / (NR/4)
        j += 1
with open ('test1.txt', 'w') as fh2:#Write that to a file so we can generate our pl
ots.
    for x in mean_scores:
        fh2.write(str(x) + '\n')
```

Sample script generating the quality score frequency data:

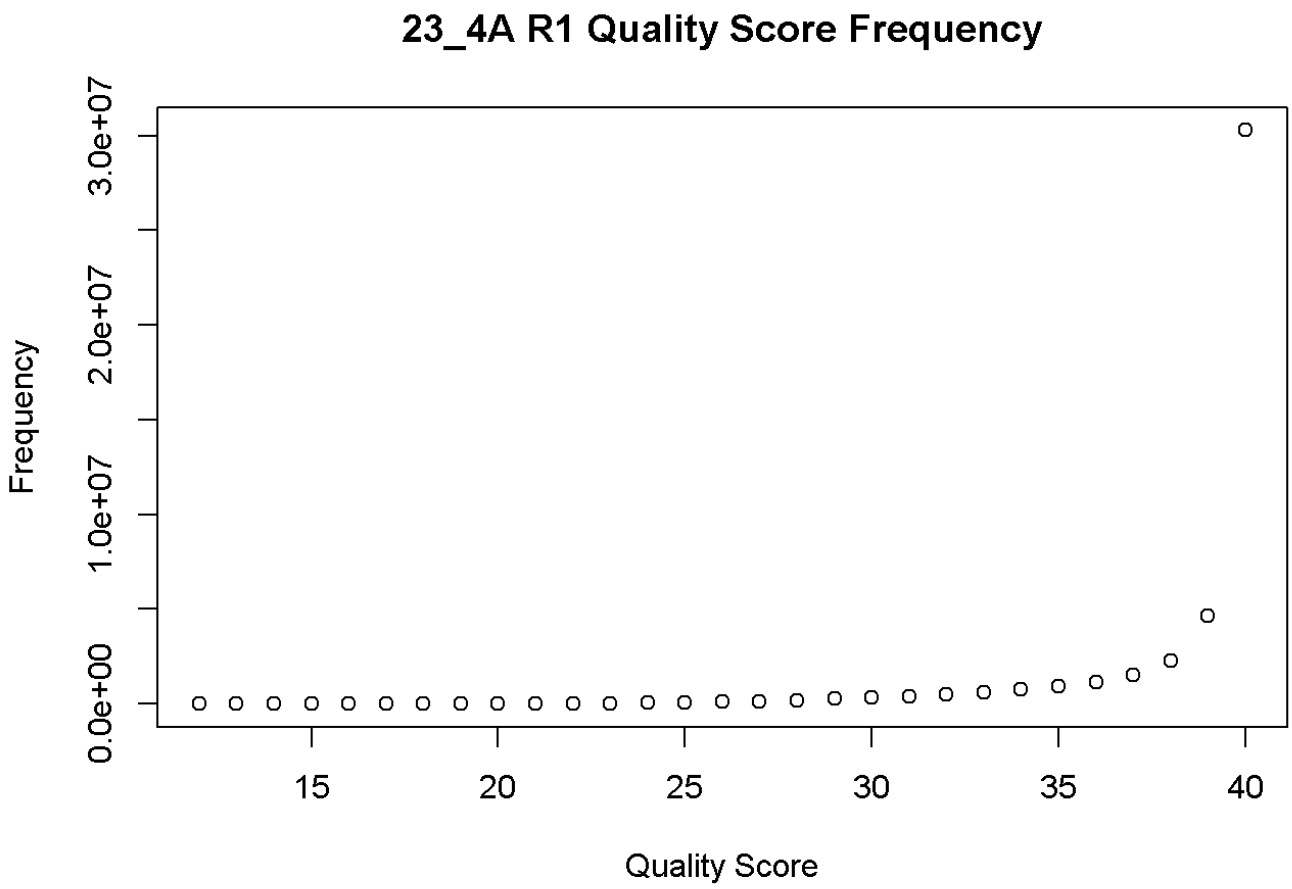
```
NR = 0 #Setting a line counter and empty dictionary
testdict = {}
with open('/home/rmeng/Bi624/SF-Seq/23_4A_control_S17_L008_R1_001.fastq') as fh:
    for line in fh:#Looping through the desired file
    readaverage = 0 #Setting our variables
        totalreadaverage = 0
        NR += 1 #Incrementing and stripping each line
        line = line.strip()
        if NR % 4 == 0: #Again only interested in quality scores in the fourth line
            for x in line: #Looping through generating the quality score for each c
haracter
                score = (ord(x)-33)
                readaverage += score #Adding each quality score to a variable
            totalreadaverage = (int(readaverage) / (len(line))) #Dividing the total
quality score of each character by the length of the line
            if totalreadaverage in testdict: #Setting a dictionary to increment fre
quencies.
                testdict[totalreadaverage] += 1
        else:
                testdict[totalreadaverage] = 1
    with open('test11.txt', 'w') as fh2: #Writing all the key-value pairs from our
dictionary to a file for analysis.
            for key in testdict:
                    fh2.write(str(key) + ':' + str(testdict[key]) + '\n')
```

Actual Plots:

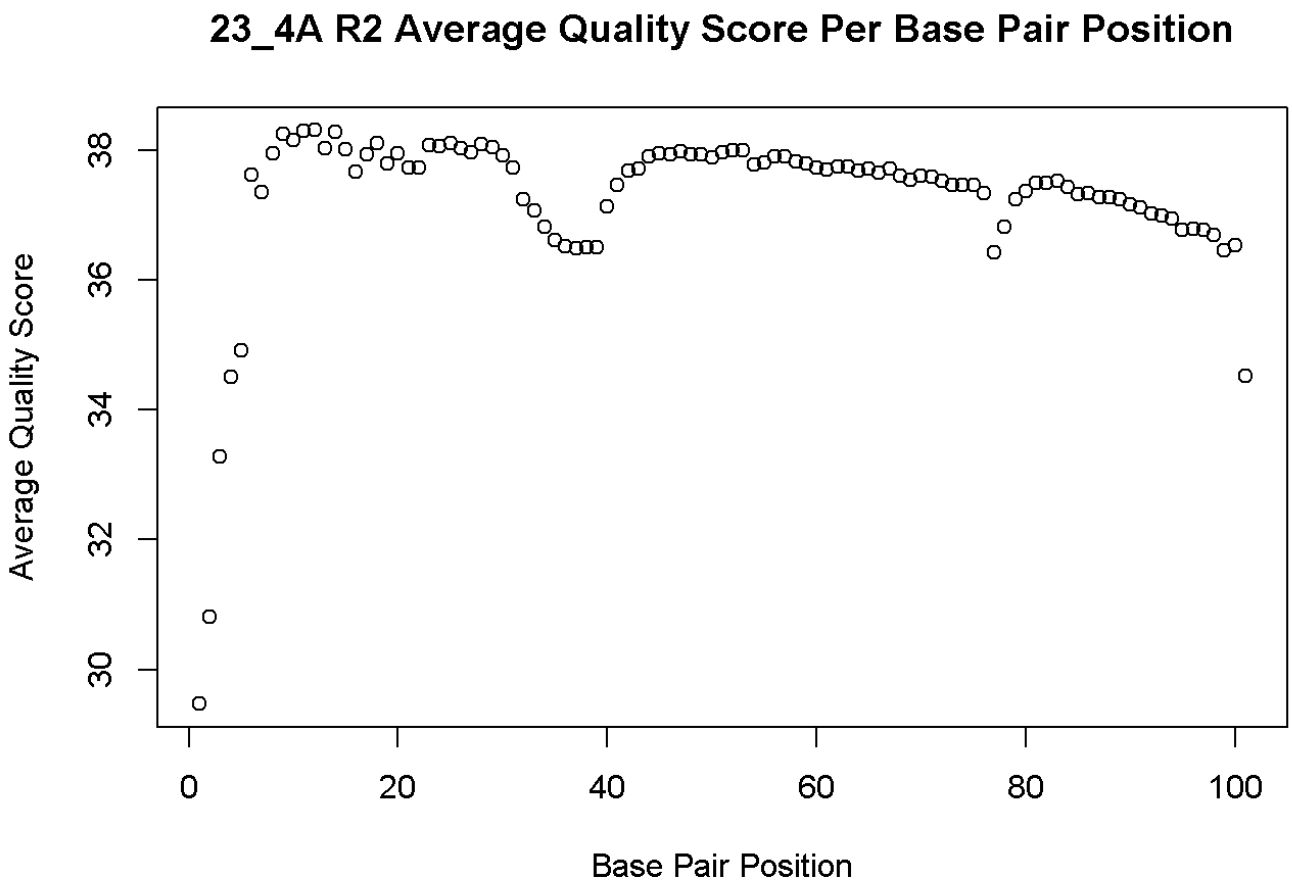
```
setwd("C:/Users/Ryan1/Documents/Bi624")

test5 <- seq(1, 101, by=1)


F23_4A_AQSPB <-read.table('test1.txt')
plot(F23_4A_AQSPB$V1~test5, main = '23_4A R1 Average Quality Score Per Base Pair Po
sition', xlab = 'Base Pair Position', ylab = 'Average Quality Score'  )
```



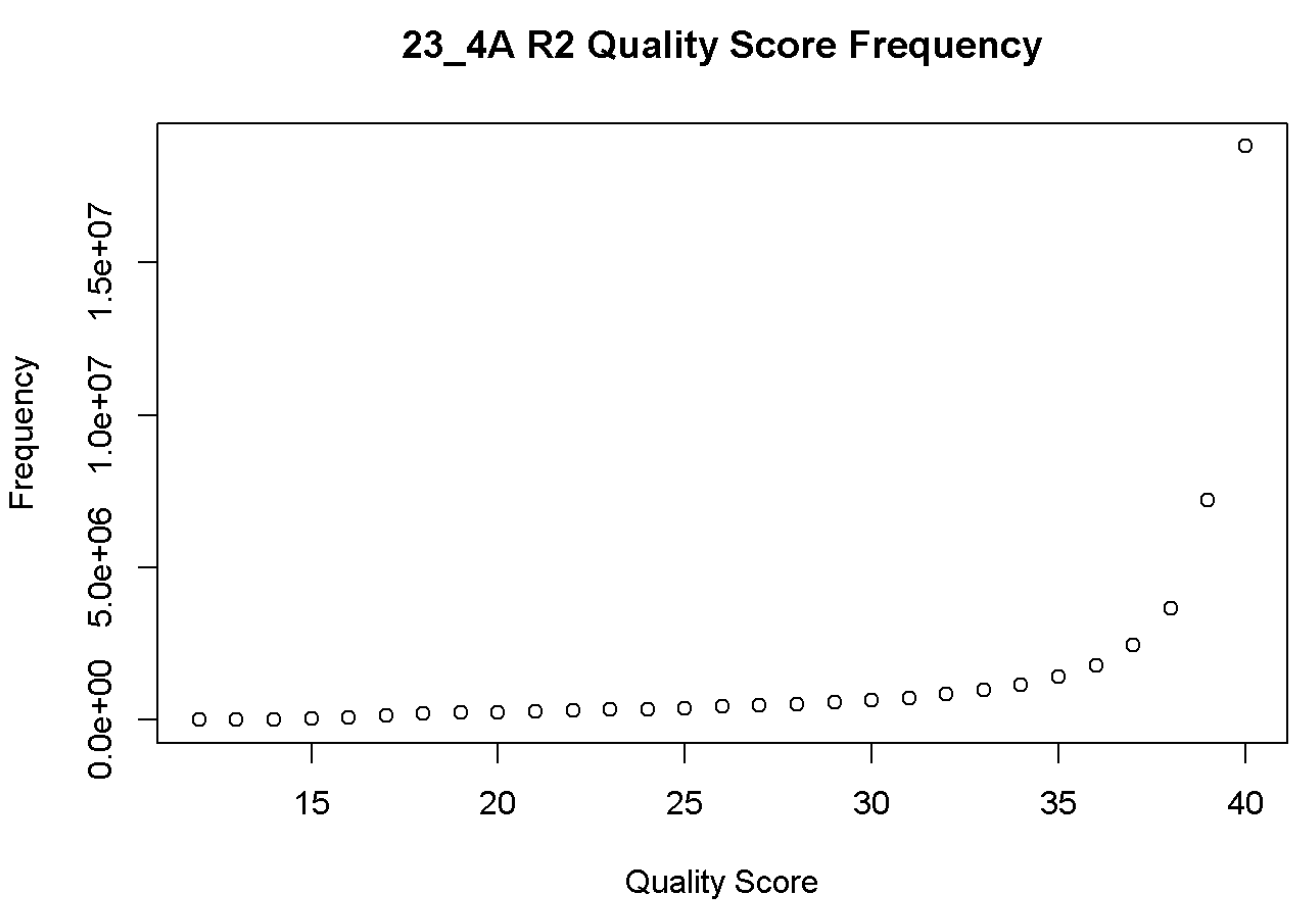

```
F23_4A_F <-read.table('test11.txt', sep = ':')
plot(F23_4A_F, main = '23_4A R1 Quality Score Frequency', xlab = 'Quality Score', y
lab = 'Frequency')
```



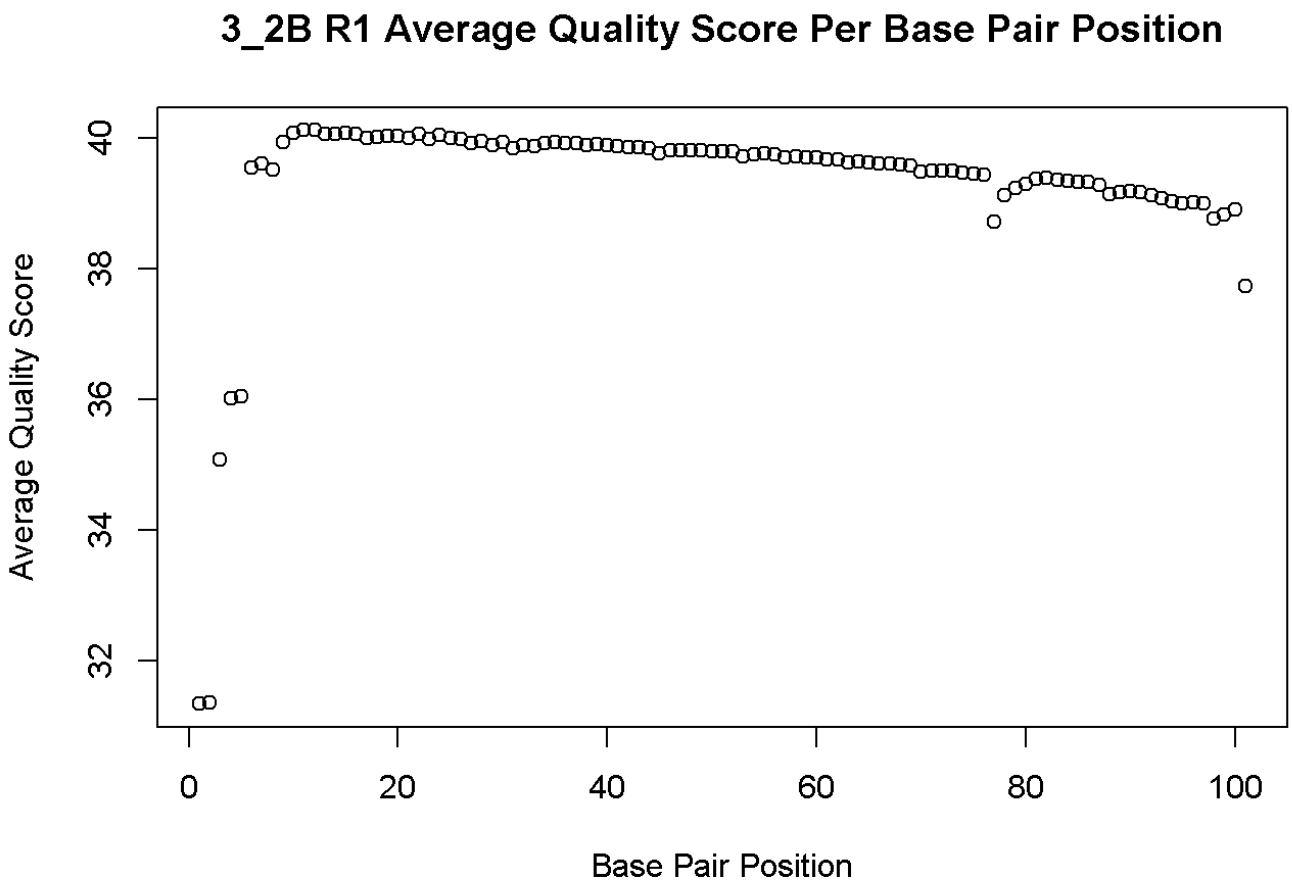

```
R23_4A_AQSPB <-read.table('test2.txt')
plot(R23_4A_AQSPB$V1~test5, main = '23_4A R2 Average Quality Score Per Base Pair Po
sition', xlab = 'Base Pair Position', ylab = 'Average Quality Score')
```



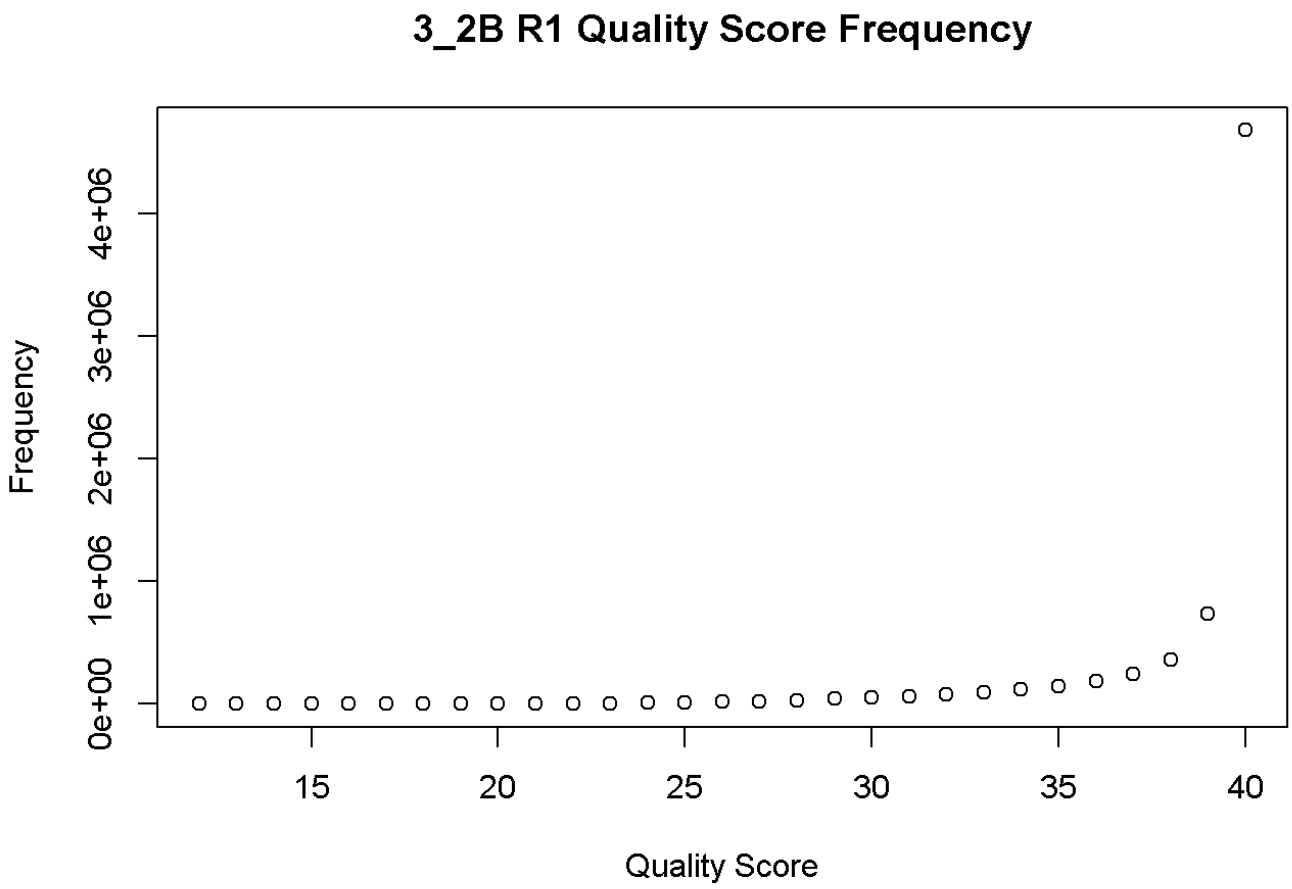

```
R23_4A_F <-read.table('test22.txt',sep = ':')
plot(R23_4A_F, main = '23_4A R2 Quality Score Frequency', xlab = 'Quality Score', y
lab = 'Frequency')
```



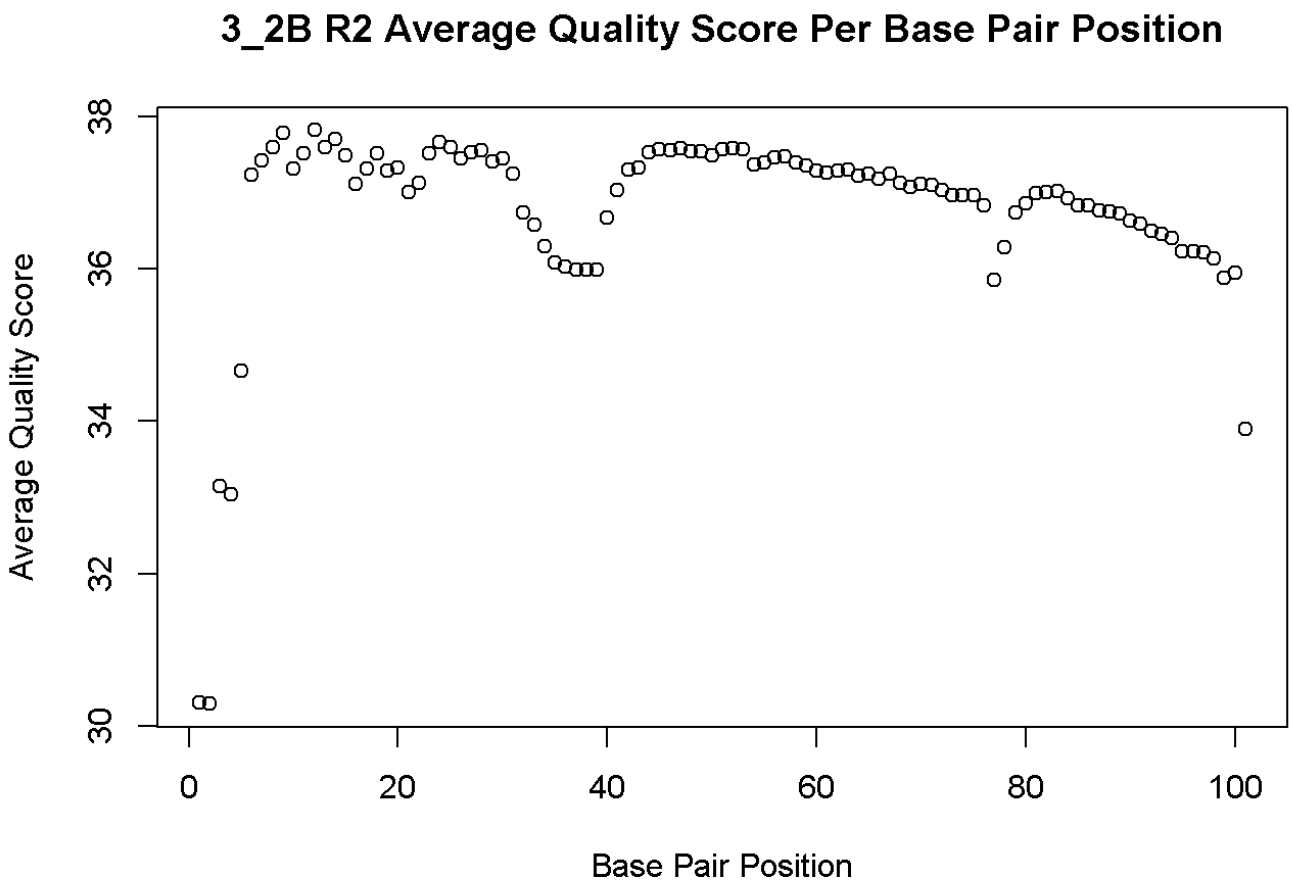

```
F3_2B_AQSPB <-read.table('test3.txt')
plot(F3_2B_AQSPB$V1~test5, main = '3_2B R1 Average Quality Score Per Base Pair Posi
tion', xlab = 'Base Pair Position', ylab = 'Average Quality Score')
```



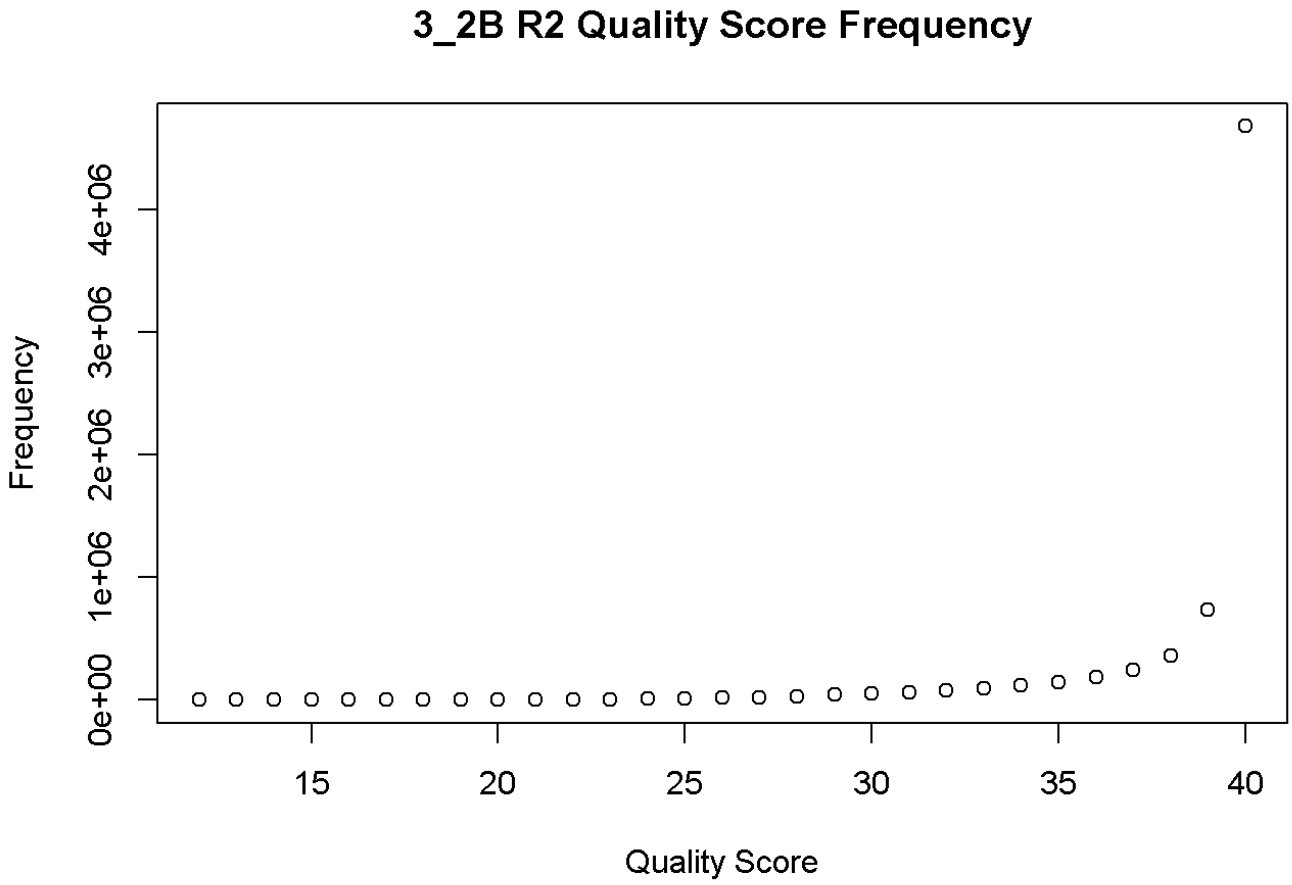

```
F3_2B_F <-read.table('test33.txt', sep = ':')
plot(F3_2B_F, main = '3_2B R1 Quality Score Frequency', xlab = 'Quality Score', yla
b = 'Frequency')
```




```
R3_2B_AQSPB <-read.table('test4.txt')
plot(R3_2B_AQSPB$V1~test5, main = '3_2B R2 Average Quality Score Per Base Pair Posi
tion', xlab = 'Base Pair Position', ylab = 'Average Quality Score')
```




```
R3_2B_F <-read.table('test44.txt', sep = ':')
plot(F3_2B_F, main = '3_2B R2 Quality Score Frequency', xlab = 'Quality Score', yla
b = 'Frequency')
```



The FASTQC quality score plots are almost identical to the plots generated using my personal scripts. Despite my plots being similar to those generated using FASTQC, FASTQC was much quicker in generating the data. This is likely because FASTQC is built on a quicker coding language (such as C++), and it is likely much more polished and efficient than my script.

Part 2 - Adaptor trimming comparison

3.

Trimmomatic is used exclusively for illumina data (both paired end and single end data) and using default parameters it removes adapters, removes leading and trailing low quality or N bases, and scans the read with a 4-base wide sliding window, cutting when the average quality per base drops below 15, dropping reads below 36 bases long.

Cutadapt finds adapter sequences, primers, poly-A tails, and other various unwanted sequence in an error-tolerant way and removes them from your data set. In addition to the aforementioned capabilities, cutadapt can also filter reads by length and do quality trimming.

Process_shortreads first begins by demultiplexing the data and checks to see if there is an error in the barcode, which can be corrected within a certain allowance. It then slides a window down the length of the read (15% of the read by default) checking the average quality score within the window. Reads with a score that drops below 90% probability of being correct are then discarded.

I chose cutadapt (cutadapt/1.14-Python-2.7.13) as the program to do my trimming.

Trimming was done on Talapas

Script:

```
#!/bin/bash
#SBATCH --job-name=Velveth      ### Job Name
#SBATCH --time=0-24:00:00       ### Wall clock time limit in Days-HH:MM:SS
#SBATCH --nodes=1               ### Node count required for the job
#SBATCH --ntasks-per-node=28     ### Number of tasks to be launched per Node
#SBATCH --partition=short       ### Partition (like a queue in PBS)

module load easybuild  icc/2017.1.132-GCC-6.3.0-2.27  impi/2017.1.132
cutadapt/1.14-Python-2.7.13

cutadapt -a AGATCGGAAGAGCACACGTCTGAACTCCAGTCAC -A AGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGT
-o out.23_4A_control_S17_L008_R1_001.fastq -p out.23_4A_control_S17_L008_R2_001.fas
tq 23_4A_control_S17_L008_R1_001.fastq 23_4A_control_S17_L008_R2_001.fastq

cutadapt -a AGATCGGAAGAGCACACGTCTGAACTCCAGTCAC -A AGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGT
-o out.3_2B_control_S3_L008_R1_001.fastq -p out.3_2B_control_S3_L008_R2_001.fastq 3
_2B_control_S3_L008_R1_001.fastq 3_2B_control_S3_L008_R2_001.fastq
```

Sanity Check:

```
cat 23_4A_control_S17_L008_R1_001.fastq | grep 'AGATCGGAAGAGCACACGTCTGAACTCCAGTCAC'
| wc -l
39965
cat 23_4A_control_S17_L008_R2_001.fastq | grep 'AGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGT'
| wc -l
46541
cat 3_2B_control_S3_L008_R1_001.fastq | grep 'AGATCGGAAGAGCACACGTCTGAACTCCAGTCAC' |
wc -l
7056
cat 3_2B_control_S3_L008_R2_001.fastq | grep 'AGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGT' |
wc -l
8157
```

Porportion of Reads Trimmed were deduced from the output files from cutadapt and confirmed using Unix commands

Example Unix Command:

```
cat out.3_2B_control_S3_L008_R2_001.fastq | awk 'NR%4==2' | awk '{ print length($0)
; }' | sort | uniq -c
wc -l 3_2B_control_S3_L008_R2_001.fastq
#The number of reads that were listed as having a length of 101 was then compared t
o the number of total reads from the original FASTA file in order to determine the
percentage of reads that were trimmed. Again this was just done as a confirmation o
f the values from the output files from cutadapt.
```

Porportion of Reads Trimmed:

23_4A_control_S17_L008_R1_001: 3.07% of reads trimmed

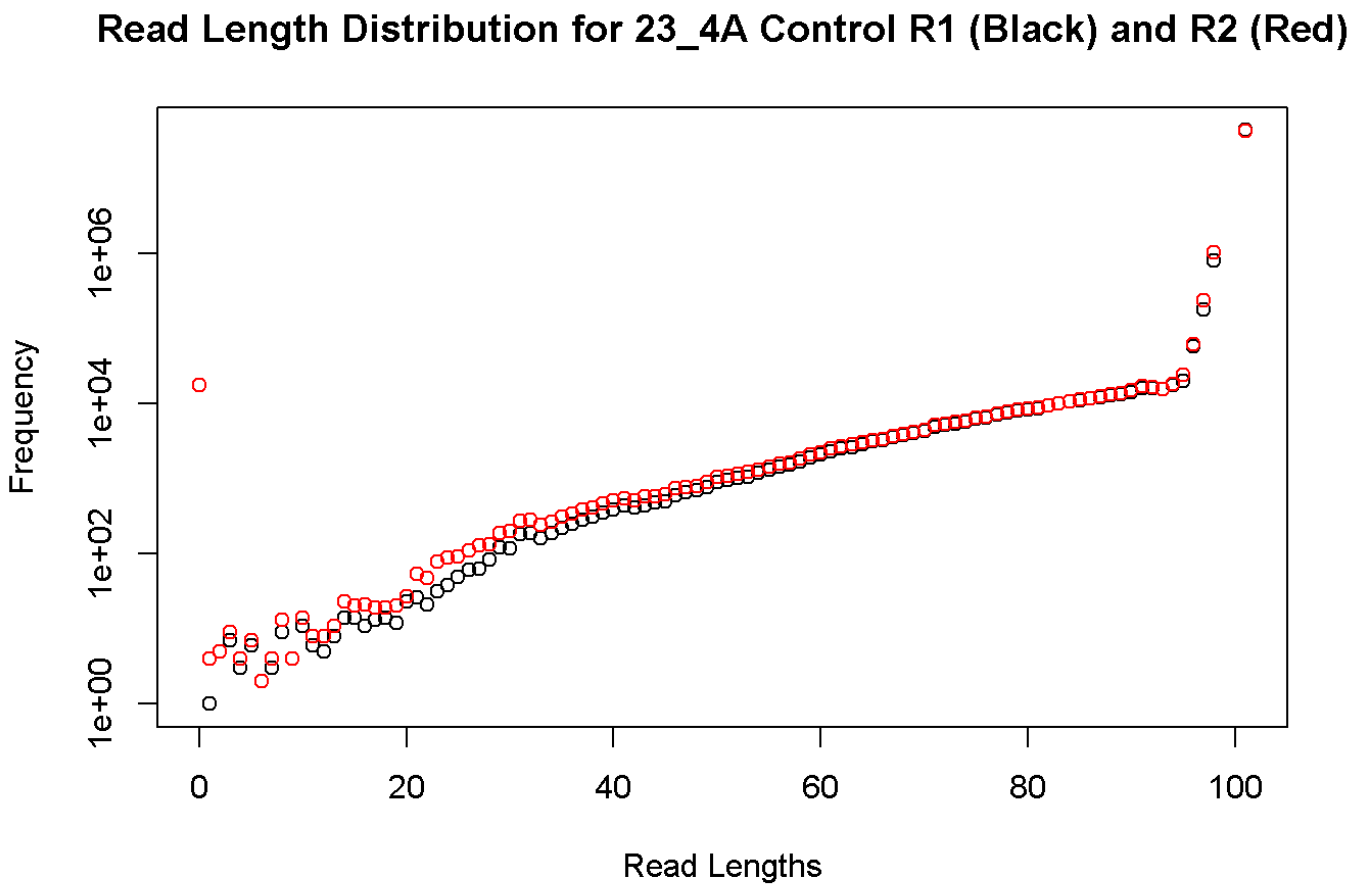
23_4A_control_S17_L008_R2_001: 3.74% of reads trimmed

3_2B_control_S3_L008_R1_001: 3.19% of reads trimmed

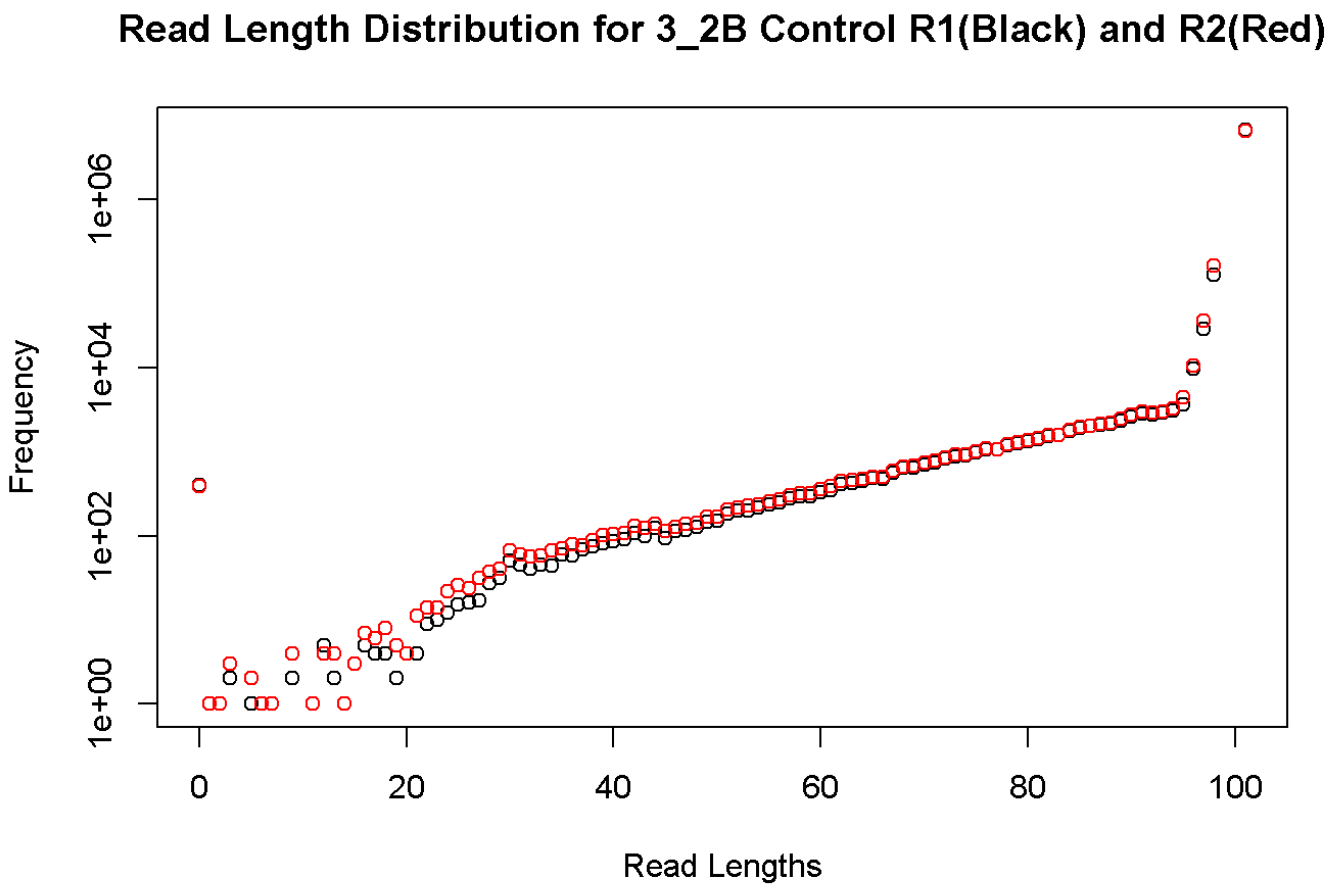
3_2B_control_S3_L008_R2_001: 3.90% of reads trimmed

4

```
test5 <-read.table('test5.txt', sep = '')
test5 <- test5[order(test5$V2),]

test6 <-read.table('test6.txt', sep = '')
test6 <- test6[order(test6$V2),]

plot(test5$V1~test5$V2, log = 'y', main = 'Read Length Distribution for 23_4A Contr
ol R1 (Black) and R2 (Red)', ylab = 'Frequency', xlab = "Read Lengths")
points(test6$V1~test6$V2, col = 'red')
```




```
test7 <-read.table('test7.txt', sep = '')
test7 <- test7[order(test7$V2),]

test8 <-read.table('test8.txt', sep = '')
test8 <- test8[order(test8$V2),]


plot(test7$V1~test7$V2, log = 'y', main = 'Read Length Distribution for 3_2B Contro
l R1(Black) and R2(Red)', ylab = 'Frequency', xlab = "Read Lengths")
points(test8$V1~test8$V2, col = 'red')
```



5

The fragment analyzer data displayed peaks at sequence length 491 bp and 427 bp for libraries 3 and 23 respectively which is consistent with our adapter trimming results. This makes sense because our sequences averages were very long which indicates not a lot of adapter contamination which was observed with the low percentage of reads being adapter trimmed. Both libraries had small peaks at 250 basepairs which would be sequences that contain adapters. Thus these small little peaks likely account for the reads that were trimmed using our trimming program.

Part 3 - rRNA reads and strand-specificity

6.

gmap-gsnap/2017-09-11 was used to generate our gmap database and used to run our gsnap commands.

My gmap database was built on an interactive node using the following command after obtaining the desired non-coding RNA fasta file from Ensembl and editing it to only include the rRNA enteries.

```
#Editing the fasta file to only include rRNA enteries
cat Mus_musculus.GRCm38.ncrna.fa | grep 'transcript_biotype:rRNA' -A 1 > gmapbuild.fa
```

gmap build script:

```
#Loading required modules
module load easybuild
module load GCC/6.3.0-2.27
module load OpenMPI/2.0.2
Module load gmap-gsnap/2017-09-11
#Building my mouse database with the file generated above form Ensembl
gmap_build -D /home/rmeng/Bi624/SF-Seq/gmap/ -d Mouse gmapbuild.fa
```

gsnap script:

```
#!/bin/bash
#SBATCH --job-name=Velveth      ### Job Name
#SBATCH --time=0-24:00:00       ### Wall clock time limit in Days-HH:MM:SS
#SBATCH --nodes=1               ### Node count required for the job
#SBATCH --ntasks-per-node=28     ### Number of tasks to be launched per Node
#SBATCH --partition=short       ### Partition (like a queue in PBS)

#Loading the required modules.
module load easybuild
module load GCC/6.3.0-2.27
module load OpenMPI/2.0.2
module load gmap-gsnap/2017-09-11

#Using my gsnap command with the paired-end format
gsnap -D /home/rmeng/Bi624/SF-Seq/gmap/Mouse -d Mouse -B 4 -m 20 -t 8 -O --split-o
utput
gsnap_out -A sam --allow-pe-name-mismatch /home/rmeng/Bi624/SF-Seq/out.23_4A_contro
l_S17_L008_R1_001.fastq /home/rmeng/Bi624/SF-Seq/out.23_4A_control_S17_L008_R2_001.
fastq

#My two gsnap commands were in seperate scripts and were exactly the same except fo
r the the files input as shown below
gsnap -D /home/rmeng/Bi624/SF-Seq/gmap/Mouse -d Mouse -B 4 -m 20 -t 8 -O --split-o
utput gsnap_out -A sam --allow-pe-name-mismatch /home/rmeng/Bi624/SF-Seq/out.3_2B_c
ontrol_S3_L008_R1_001.fastq /home/rmeng/Bi624/SF-Seq/out.3_2B_control_S3_L008_R2_00
1.fastq
```

Calculating the proportion of reads that mapped back for each pair of my files:

```
wc -l out.23_4A_control_S17_L008_R1_001.fastq
177213048 out.23_4A_control_S17_L008_R1_001.fastq
```

```
wc -l out.23_4A_control_S17_L008_R2_001.fastq
177213048 out.23_4A_control_S17_L008_R2_001.fastq
```

177213048/4 = 44,303,262*2 = 88,606,524

```
cat gsnap_out.nomapping | grep -v '^@' | wc -l
88500923
```

1-(88,500,923/88,606,524) = **0.11917% of reads likely came from rRNA for 23_4A Control**

```
wc -l out.3_2B_control_S3_L008_R1_001.fastq
27494036 out.3_2B_control_S3_L008_R1_001.fastq
```

```
wc -l out.3_2B_control_S3_L008_R2_001.fastq
27494036 out.3_2B_control_S3_L008_R2_001.fastq
```

27494036 / 4 = 6,873,509 * 2 = 13,747,018

```
cat gsnap_out.nomapping | grep -v '^@' | wc -l
13744583
```

1-(13,744,583/13,747,018) = **0.0177% of reads likely came from rRNA for 3_2B control**

7

To determine strand specificity, I grepped for 20 consecutive A's and T's in each file and have shown the results both numerically and graphically.

Sample Command:

```
awk 'NR % 4 == 2' 3_2B_control_S3_L008_R1_001.fastq | grep 'TTTTTTTTTTTTTTTTTTT' |
wc -l

awk 'NR % 4 == 2' 3_2B_control_S3_L008_R1_001.fastq | grep 'AAAAAAAAAAAAAAAAAAA' |
wc -l
```

Results:

23_4A_R1 Poly A 21152

23_4A_R2 Poly A **45030**

23_4A_R1 Poly T **141142**

23_4A_R2 Poly T 46005

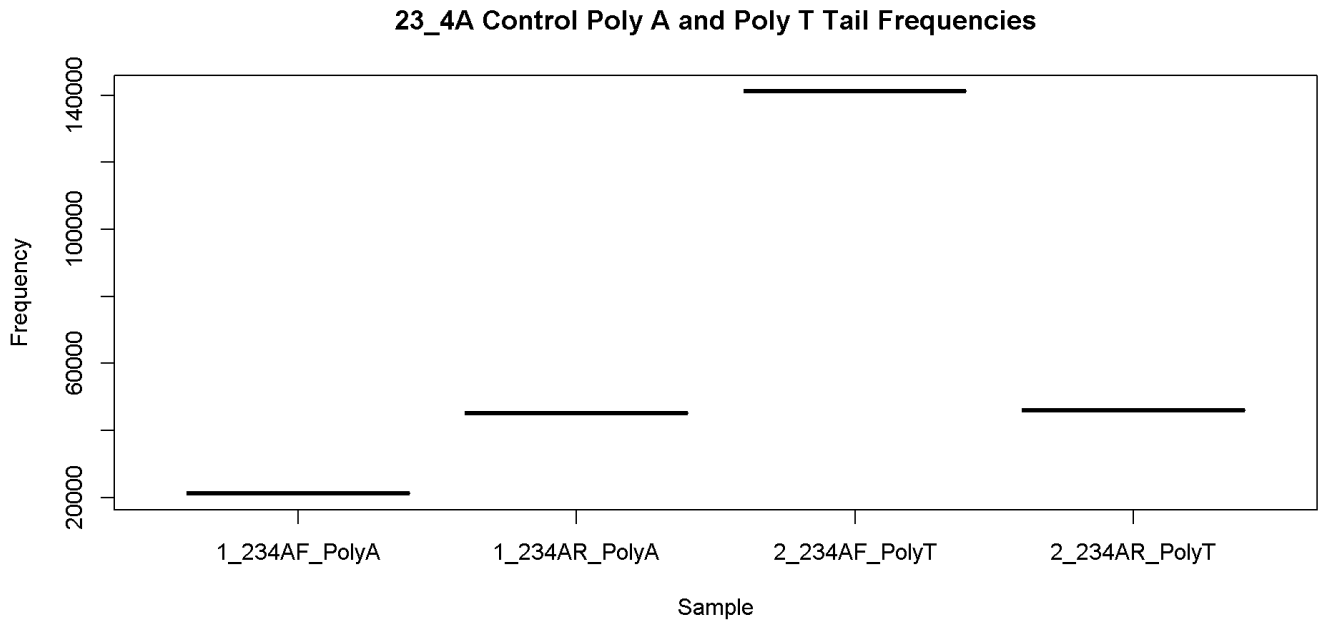
3_2B_R1 Poly A 3368

3_2B_R2 Poly A **7568**

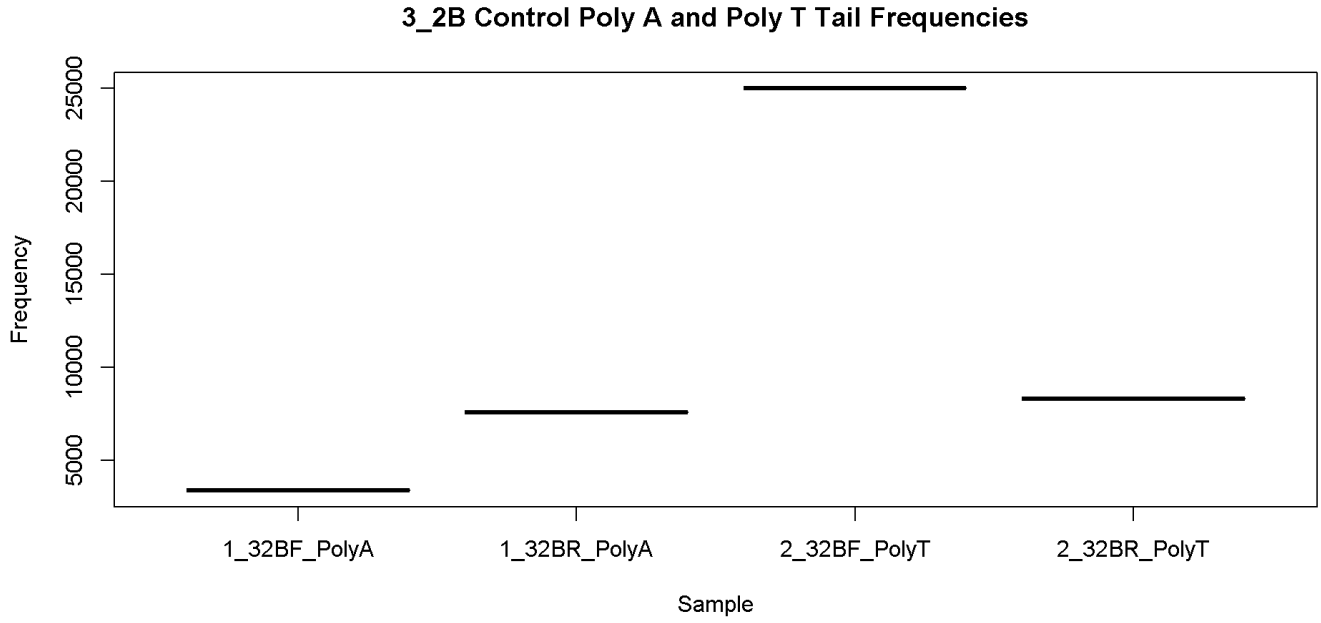
3_2B_R1 Poly T **24968**

3_2B_R2 Poly T 8278

```
test100 <-read.table('test100.txt', sep = '')
plot(test100$V2~test100$V1, main = '23_4A Control Poly A and Poly T Tail Frequenci
es', xlab = 'Sample', ylab = 'Frequency')
```




```
test101 <-read.table('test101.txt', sep = '')
plot(test101$V2~test101$V1, main = '3_2B Control Poly A and Poly T Tail Frequencie
s', xlab = 'Sample', ylab = 'Frequency')
```



The stark differences between the two files in terms of strands of consecutive T's and consecutive A's is indicitive of strandness.