QAA

Keenan Raleigh

2022-09-06

Part 1

FASTQC commands:

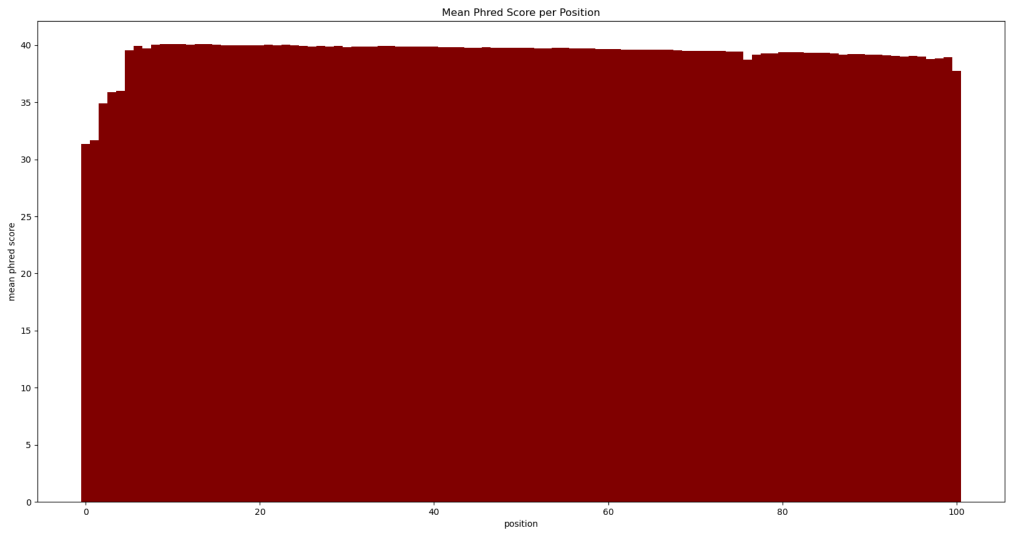
```
fastqc 1_2A_control_S1_L008_R1_001.fastq.gz -o /projects/bgmp/kraleigh/bi623
fastqc 1_2A_control_S1_L008_R2_001.fastq.gz -o /projects/bgmp/kraleigh/bi623
fastqc 24_4A_control_S18_L008_R1_001.fastq.gz -o /projects/bgmp/kraleigh/bi623
fastqc 24_4A_control_S18_L008_R2_001.fastq.gz -o /projects/bgmp/kraleigh/bi623
```

The script I used to make the per base quality histograms is phred_scores.py.
All scripts used will be uploaded to the github repository in scripts.

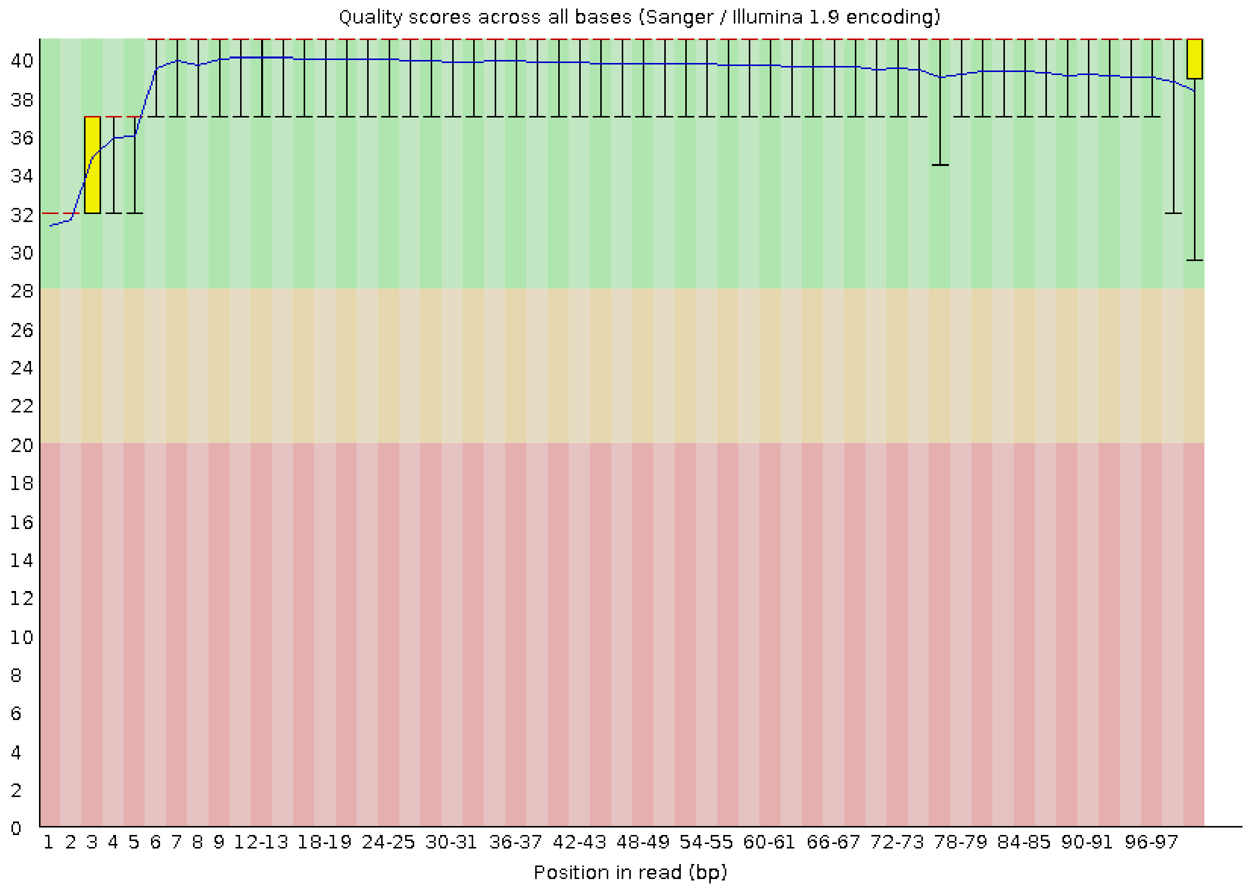
per base quality:

Library 1 read 1:

```
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/Q-scores/1_R1_hist.png", error = FALSE)
```

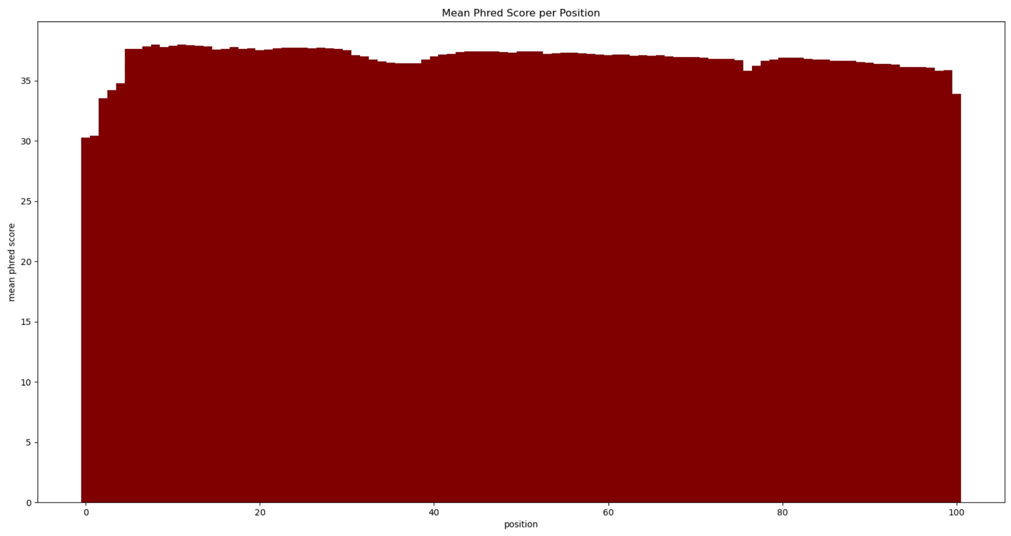


```
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/Q-scores/1_r1_per_base_quality.png", error
```

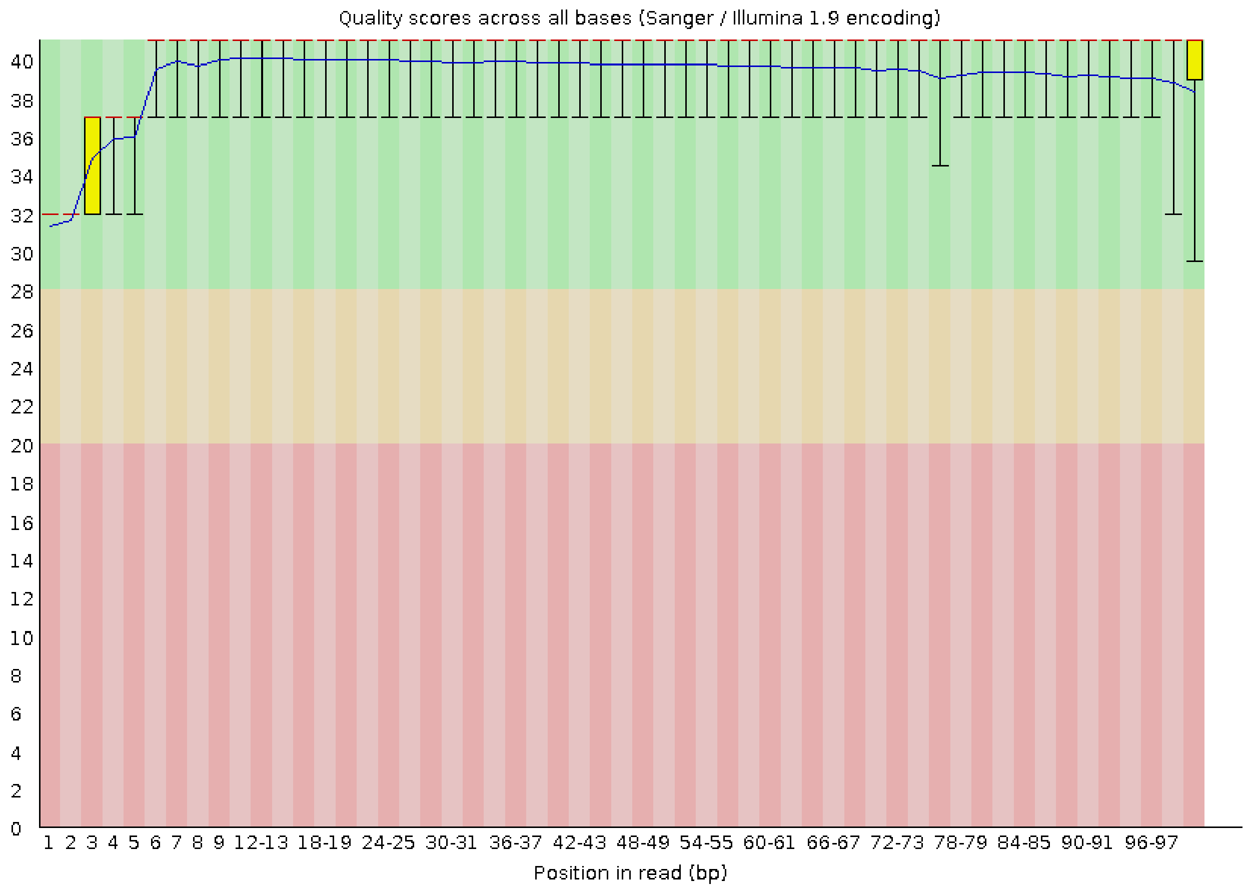


Library 1 read 2:

```
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/Q-scores/1_R2_hist.png", error = FALSE)
```

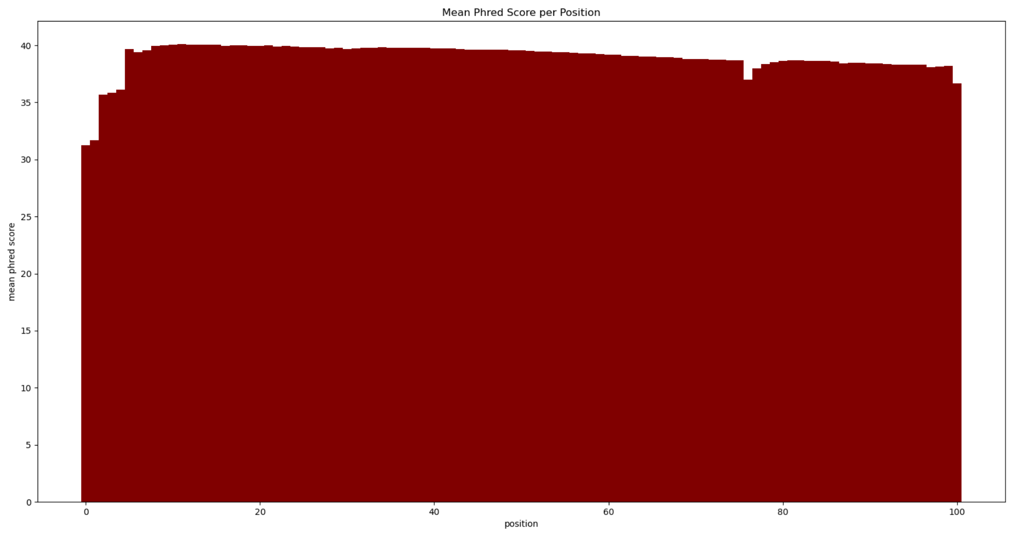


```
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/Q-scores/1_r1_per_base_quality.png", error =
```

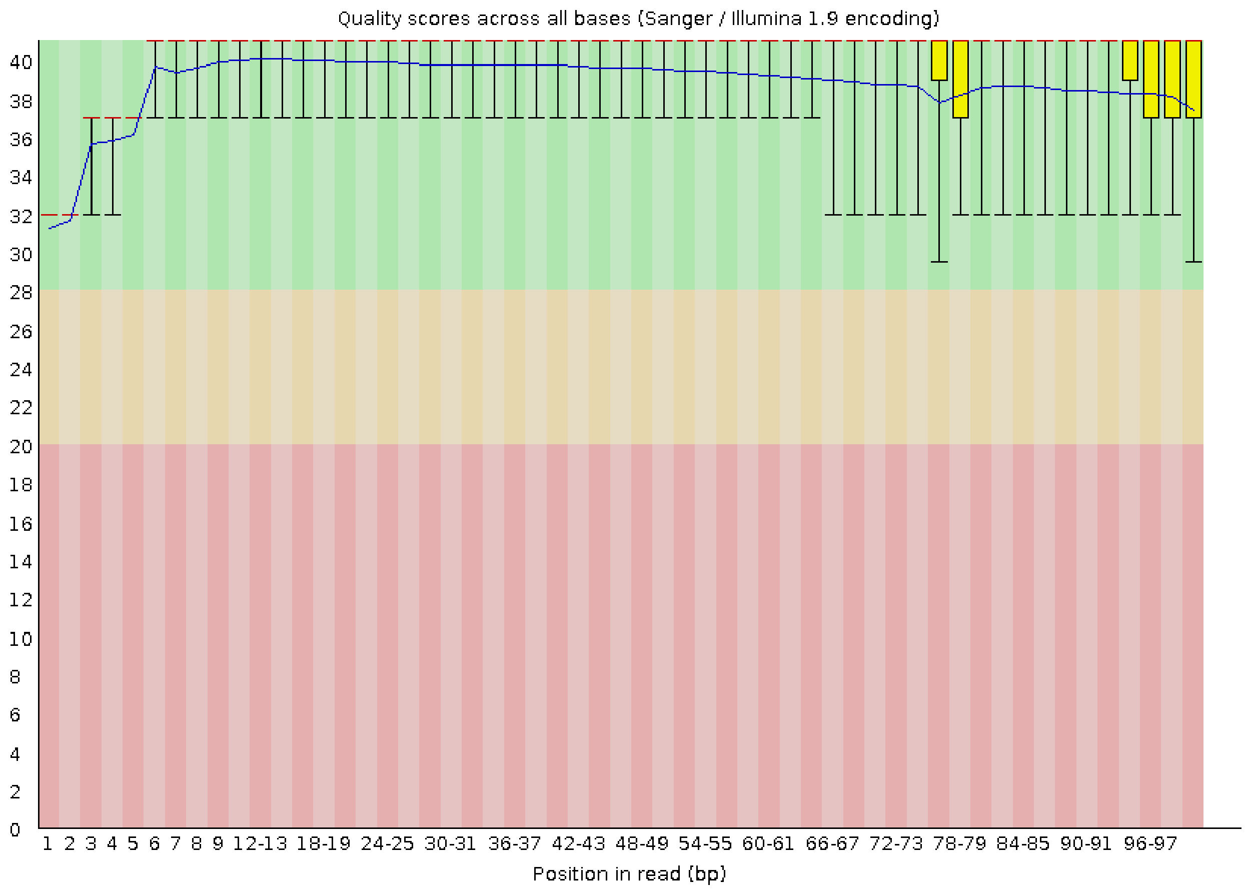


Library 24 read 1:

```
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/Q-scores/24_R1_hist.png", error = FALSE)
```

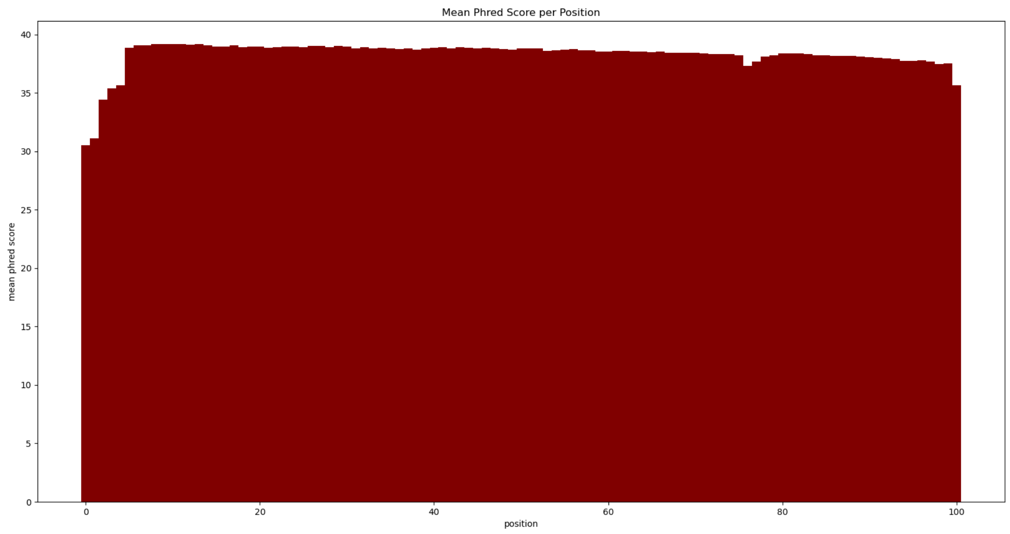


```
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/Q-scores/24_r1_per_base_quality.png", error
```

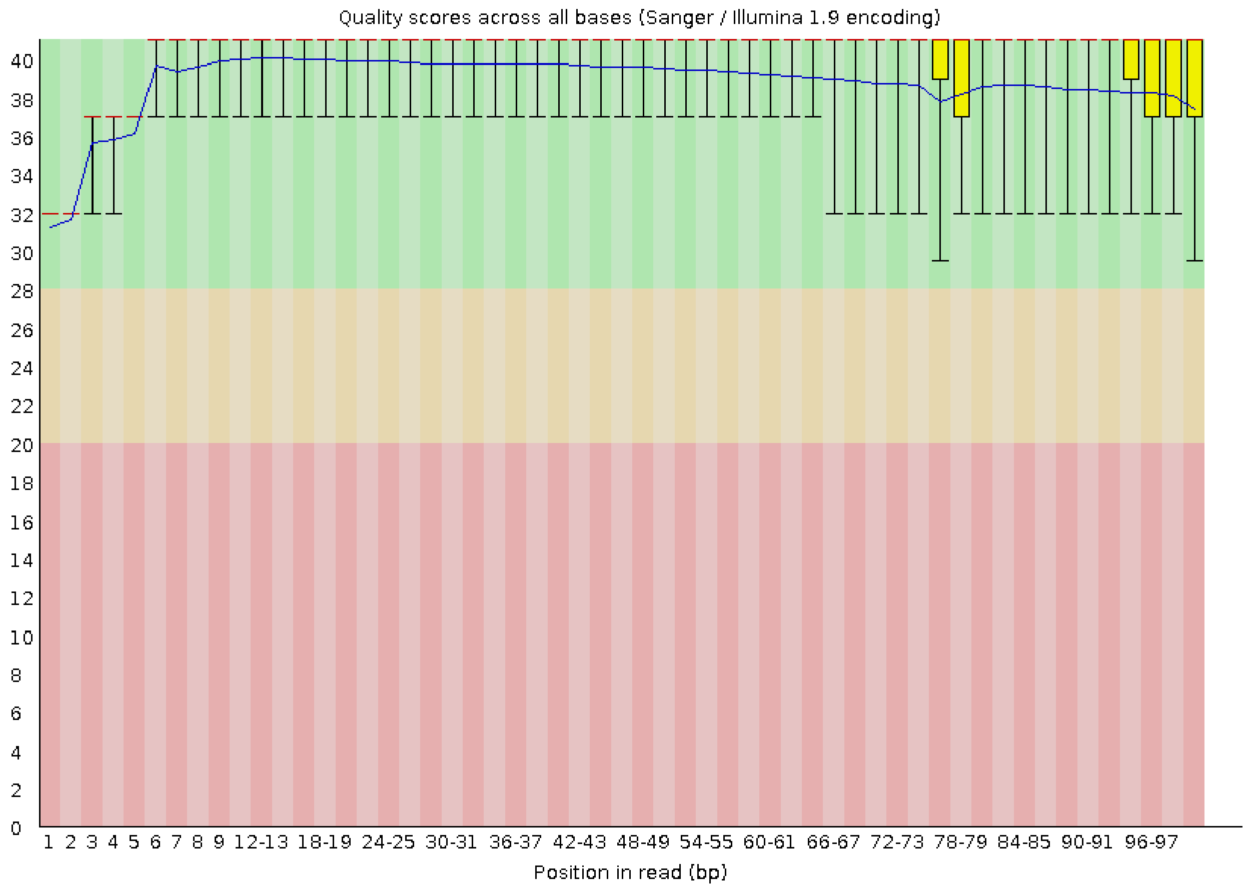


Library 24 read 2:

```
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/Q-scores/24_R2_hist.png", error = FALSE)
```



```
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/Q-scores/24_r1_per_base_quality.png", error
```

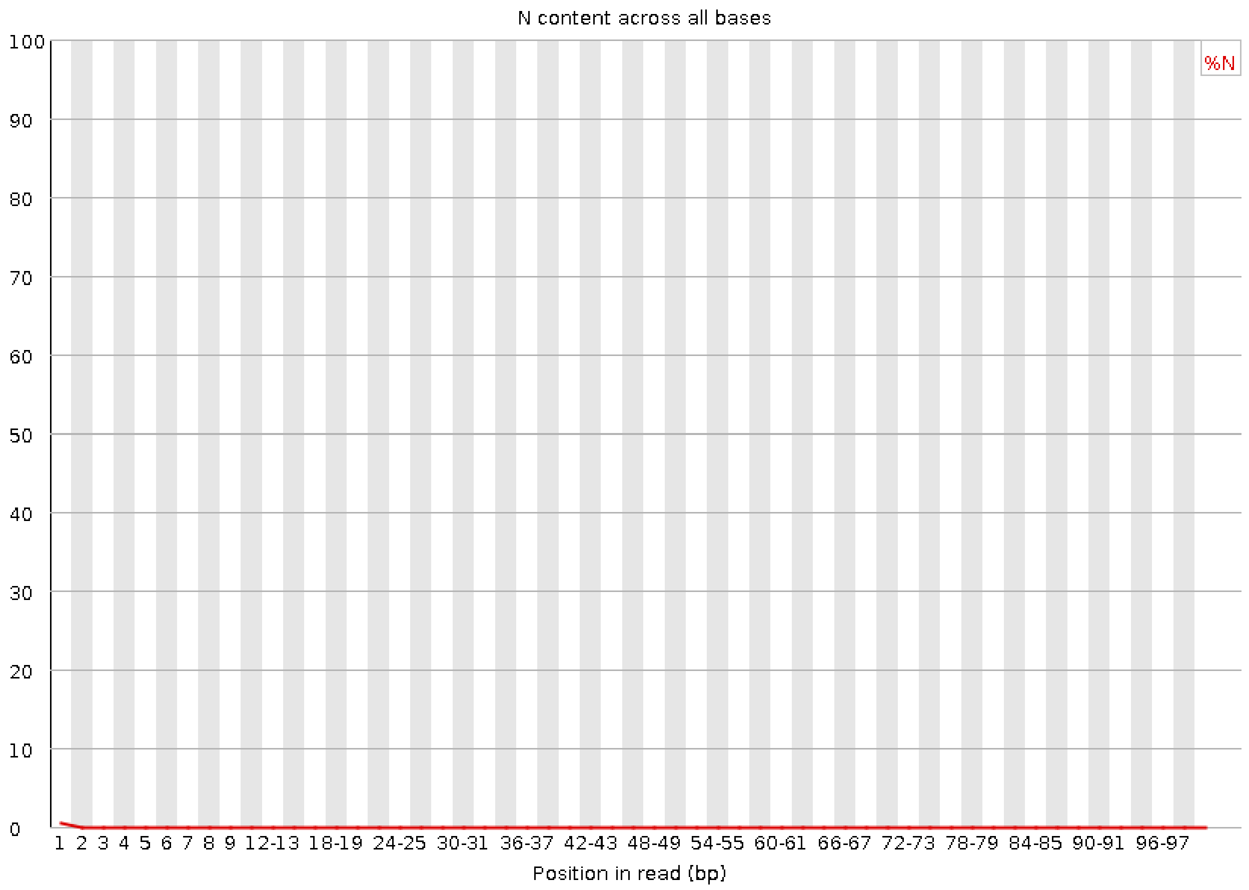


The data from our script matches the data from fastqc track very closely they all show the same lower quality scores at the beginning and the end as well as a dip around the 78th position. It was striking how much faster the fastqc program ran, especially considering how much more information it was compiling but I suspect this is probably because there was a team of professional developers that made the fastqc program as efficiently as possible while my script was written by a student with about a month of experience.

Per base n content:

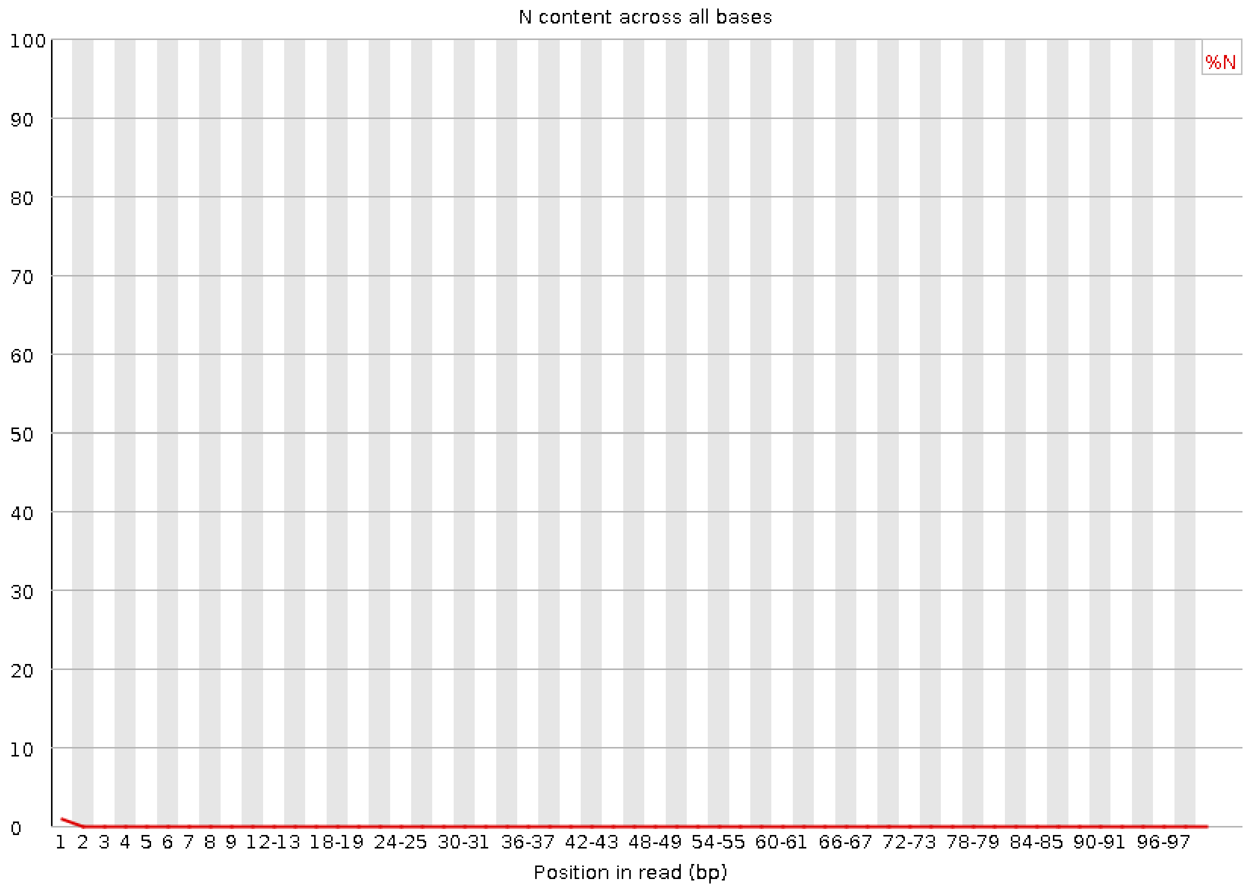
Library 1 read 1

```
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/1_r1_per_base_n_content.png", error = FALSE
```



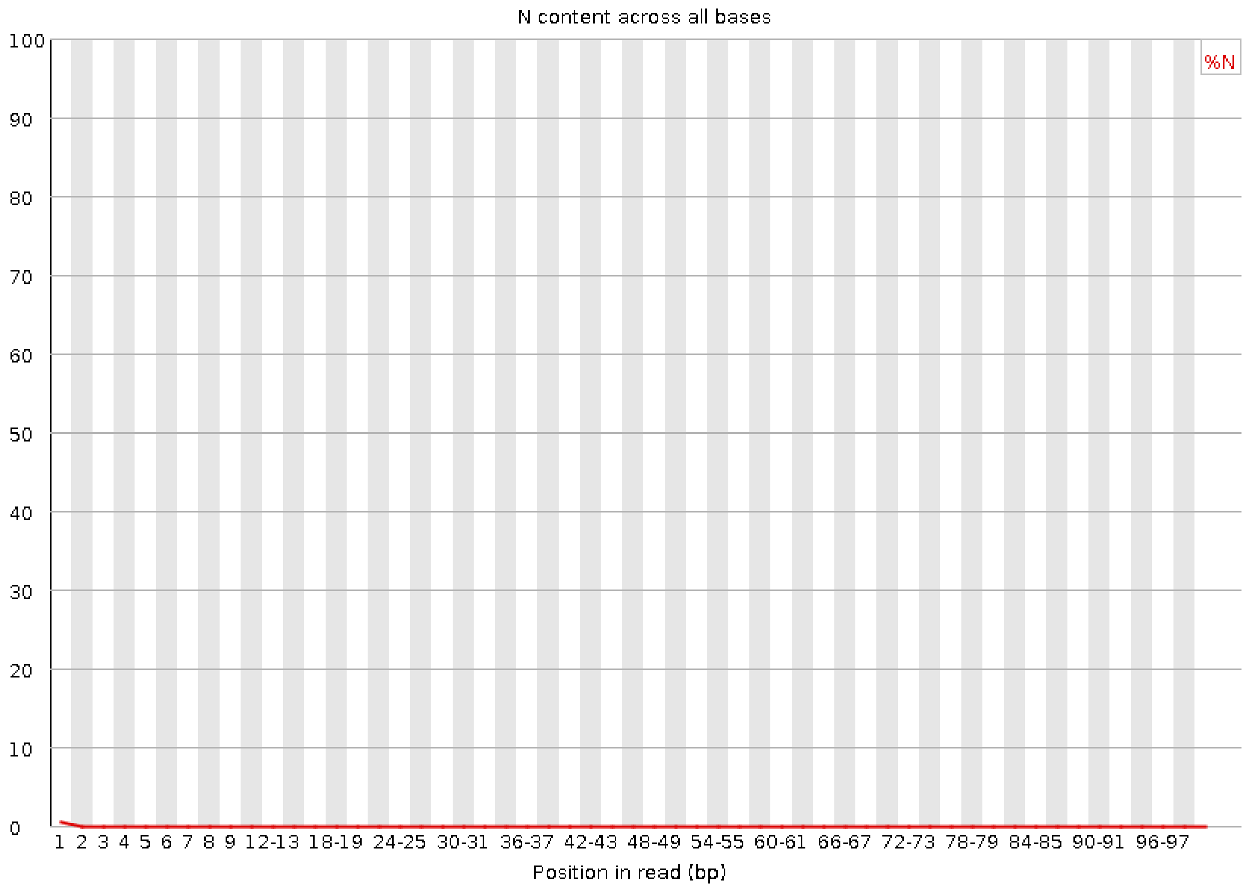
Library 1 read 2:

```
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/1_r2_per_base_n_content.png", error = FALSE
```



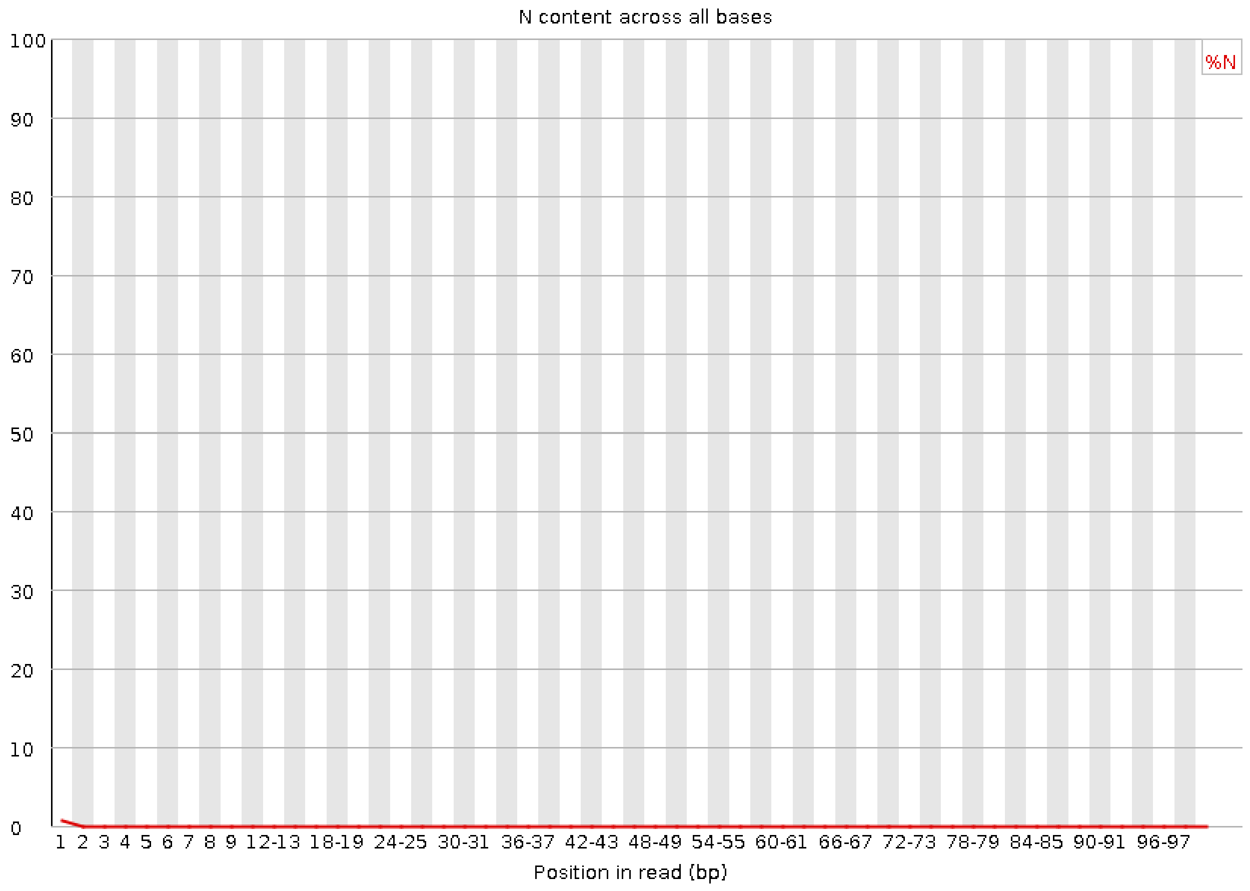
Library 24 read 1:

```
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/24_r1_per_base_n_content.png", error = FALSI
```



Library 24 read 2:

```
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/24_r2_per_base_n_content.png", error = FALS
```



The per base n content matches our quality scores very well, the only noticeable tick on the graph matches the lowest quality scores at the first and second positions which is expected in Illumina sequencing.

Overall quality of the reads: I believe the overall quality of libraries 1 and 24 are pretty good. Fastqc raises concerns at a couple of the graphs but their concerns match what we would expect in an Illumina sequence. there does seem to be a band of slightly problematic data in the per tile sequence quality from the 2201 to 2216 tiles for the second reads but we do expect to have lower quality from the second reads in general. There is also a warning at the kmer content but this is also expected.

Part 2

commands used for cutadapt: cutadapt -a AGATCGGAAGAGCACACGTCTGAACTCCAGTCA -o ../../../kraleigh/bi623/1_r1_trmd.fastq 1_2A_control_S1_L008_R1_001.fastq.gz
cutadapt -a AGATCGGAAGAGCACACGTCTGAACTCCAGTCA -o ../../../kraleigh/bi623/24_r1_trmd.fastq.gz 24_4A_control_S18_L008_R1_001.fastq.gz
cutadapt -aAGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGT -o ../../../kraleigh/bi623/24_r2_trmd.fastq.gz 24_4A_control_S18_L008_R2_001.fastq.gz
cutadapt -aAGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGT -o ../../../kraleigh/bi623/1_r2_trmd.fastq.gz 1_2A_control_S1_L008_R2_001.fastq.gz

commands used to make a text file with the lengths and counts of each read: zcat 1_r1_trmd.fastq.gz | grep -v @@ | grep -A1 @ | grep -v @ | grep -v - | awk '{print length ($0)}' | sort | uniq

-c | sort -n > 1_r1_len.txt
zcat 1_r2_trmd.fastq.gz | grep -v @@ | grep -A1 @ | grep -v @ | grep -v - | awk '{print length ($0)}' | sort | uniq -c | sort -n > 1_r2_len.txt
zcat 24_r1_trmd.fastq.gz | grep -v @@ | grep -A1 @ | grep -v @ | grep -v - | awk '{print length ($0)}' | sort | uniq -c | sort -n > 24_r1_len.txt
zcat 24_r2_trmd.fastq.gz | grep -v @@ | grep -A1 @ | grep -v @ | grep -v - | awk '{print length ($0)}' | sort | uniq -c | sort -n > 24_r2_len.txt

cutadapt results: I wrote a script that outputted the proportion of reads that were trimmed and untrimmed called trimmed_or_not.py uploadedto the github repository in scripts

library 1 read 1: ./trimmed_or_not.py -t 1_r1_len.txt
468835 trimmed reads, 8009024 untrimmed reads
5.53011084520278 percent of reads were trimmed

library 1 read 2: ./trimmed_or_not.py -t 1_r2_len.txt
537308 trimmed reads, 7940551 untrimmed reads
6.337779385101829 percent of reads were trimmed

library 24 read 1: ./trimmed_or_not.py -t 24_r1_len.txt
335742 trimmed reads, 10180132 untrimmed reads
3.1927160785684574 percent of reads were trimmed

library 24 read 2: ./trimmed_or_not.py -t 24_r2_len.txt
417709 trimmed reads, 10098165 untrimmed reads
3.9721757792077006 percent of reads were trimmed

Trimmomatic commands and results

Trimmomatic commands: trimmomatic PE 1_r1_trmd.fastq.gz 1_r2_trmd.fastq.gz 1_r1_trmmd.fastq.gz 1_r1_untrmmd.fastq.gz 1_r2_trmmd.fastq.gz 12_untrmmd.fastq.gz LEADING:3 TRAILING:3 SLIDINGWINDOW:5:15 MINLEN:35

trimmomatic PE 24_r1_trmd.fastq.gz 24_r2_trmd.fastq.gz 24_r1_trmmd.fastq.gz 24_r1_untrmmd.fastq.gz 24_r2_trmmd.fastq.gz 24_r2_untrmmd.fastq.gz LEADING:3 TRAILING:3 SLIDINGWINDOW:5:15 MINLEN:35

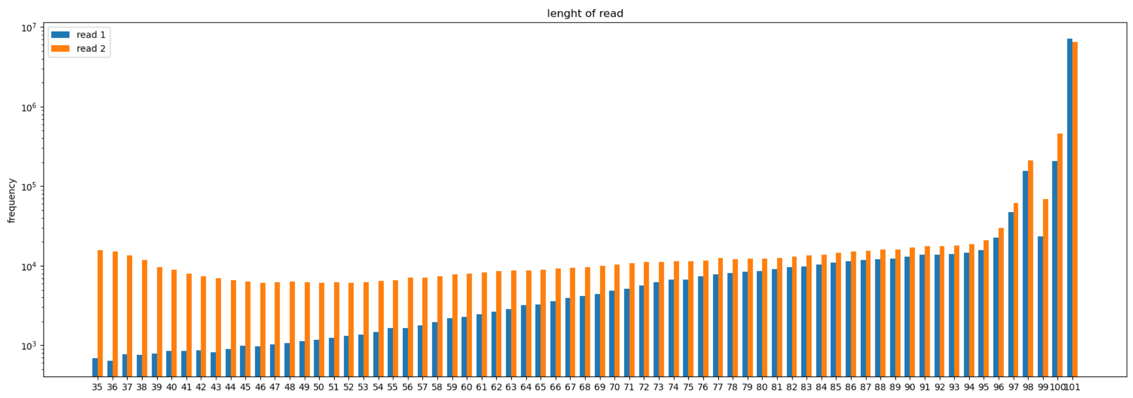
Trimming results:

Commands to make text files with read lengths and counts: zcat 1_r1_trmmd.fastq.gz | grep -v @@ | grep -A1 @ | grep -v @ | grep -v - | awk '{print length ($0)}' | sort | uniq -c | sort -n > 1_r1_rd_len.txt
zcat 1_r2_trmmd.fastq.gz | grep -v @@ | grep -A1 @ | grep -v @ | grep -v - | awk '{print length ($0)}' | sort | uniq -c | sort -n > 1_r2_rd_len.txt
zcat 24_r1_trmmd.fastq.gz | grep -v @@ | grep -A1 @ | grep -v @ | grep -v - | awk '{print length ($0)}' | sort | uniq -c | sort -n > 24_r1_rd_len.txt
zcat 24_r2_trmmd.fastq.gz | grep -v @@ | grep -A1 @ | grep -v @ | grep -v - | awk '{print length ($0)}' | sort | uniq -c | sort -n > 24_r2_rd_len.txt

I wrote a python script to generate a histogram of the read lengths for r1 and r2 together called histogram.py.

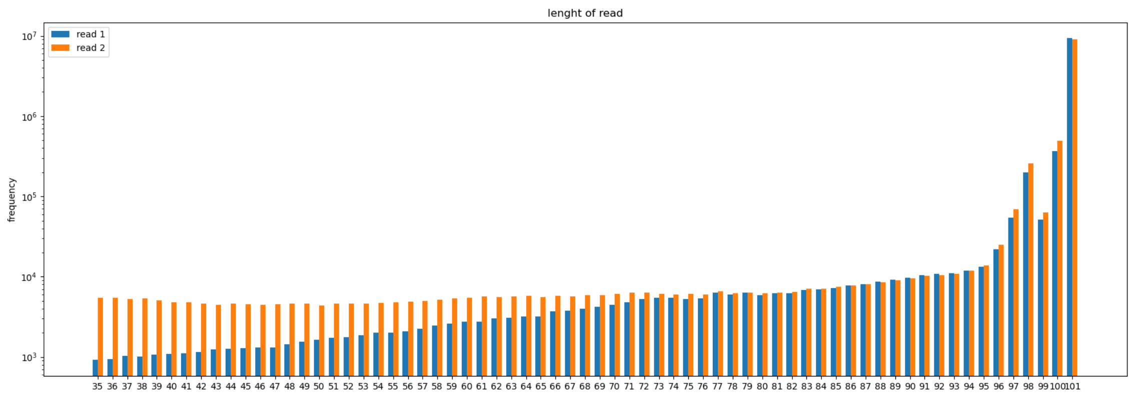
library 1:

```r
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/1_trim_histogram.png", error = FALSE)
```



library 24:

```r
knitr::include_graphics("C:/Users/ralei/Desktop/bioinfo/QAA/24_trim_histogram.png", error = FALSE)
```



The results of our trimming clearly show increased trimming in second reads, this is expected because we saw that the second reads had much lower quality than the first reads. We did not see a clear difference in trimming from cutadapt in the two reads, the first reads are slightly less trimmed but with a sample size of 2 the difference is too small to draw a meaningful conclusion. This does make sense because the presence of the adapter sequence in the read is not going to change based on when the sample was read while the quality of the base calls does decrease with time.

Part 3

Alignment and strandedness

We downloaded a mouse genome FASTA file and GFT file from ensemble 107 to run alignment software on STAR.

We created a genome database with the bash script starstuff.sh and aligned them with starstuff2.sh.

Then I wrote a script to count to mapped and unmapped reads based on bitflags called mapped_unmapped.py.

Alignment results from mapped_unmapped: ./mapped_unmapped.py -f alignment_SAM_1/Aligned.out.sam

```
{'mapped': 15627441, 'unmapped': 306498}
./mapped_unmapped.py -f alignment_SAM_24/Aligned.out.sam
{'mapped': 19780644, 'unmapped': 711046}
```

htseq stuff

I used htseq to count the mapped and unmapped reads, the bash script I used is called htseq_count.sh.

htseq_count.sh results

```
./read_count.py -f lib_1_yes.txt
there are 312665 mapped reads and 7653678 unmapped reads out of 7966343 reads
3.924824728234775 % mapped reads
fin_libs]$ ./read_count.py -f lib_1_reverse.txt
there are 6877200 mapped reads and 1089143 unmapped reads out of 7966343 reads
86.32819350108325 % mapped reads
fin_libs]$ ./read_count.py -f lib_24_yes.txt
there are 356658 mapped reads and 9888774 unmapped reads out of 10245432 reads
3.4811416443933254 % mapped reads
fin_libs]$ ./read_count.py -f lib_24_reverse.txt
there are 8377321 mapped reads and 1868111 unmapped reads out of 10245432 reads
81.76640087016341 % mapped reads
```

strandedness

The results of the htseq count shows that the libraries are stranded. We can tell because the yes and reverse results are clearly different in percentage of mapped reads. This is because the yes option is mapping to the positive strand while reverse is mapping to the negative strand and if the library was not stranded the two counts would have resulted in roughly even read counts.