**Neuro-genomics - class 2**

Last week we introduced the concept of turning the RNA content of a tissue sample into a vector of length ~20,000. This vector represents the expression of each one of the genes in the tissue. Today, we will talk about how this vector is constructed. Next week we will discuss how to compare this vector between different kinds of tissues, or between several conditions of the same tissue type (for example, disease and healthy conditions).

We will not try to present an exhaustive account on all the biochemical and analysis steps that are required to create the aforementioned vector. Rather, we will focus on steps that highlight important aspects of the resulting data. The general motivation is simple: if we want to quantify the data, we need to understand what possible artifacts are involved in its creation.

Question: most sequencing attempts are aimed at RNA, and not DNA, why is that?
Answer: It is cheaper to sequence RNA as the DNA is much longer. Moreover, the DNA is expected to be more or less the same in all tissues (exceptions: tumors and probably neurons), whereas the RNA is different in every tissue. The RNA serves as a proxy for proteins, and is expected to change constantly either because of the environment (example, light is reaching the retina, smoke is reaching our lungs), or because of internal changes (for example, the circadian clock or the cell cycle). Therefore, if we want to study disease states, or to examine the effect of a factor on a tissue, we will usually use RNAseq and not DNAseq.

There are three main steps in the creation of the aforementioned vector:

1) Isolation of RNA molecules from the tissue
2) Amplification and sequencing of RNA molecules
3) Alignment of RNA sequences against the genes/genome

**Step 1: Isolation of RNA molecules from the tissue**

The tissue is usually destroyed by physical means (mechanical force, sound waves etc). The RNA is then isolated either using its physical properties (negative change, size etc) or using the fact that mRNA ends with poly(A) tail, and therefore beads with poly(T) will bind the mRNA.

Typically only the mRNA are sequenced, using poly(A) enrichment of the mRNA as described above. However, non-poly(A) transcripts can also be sequenced, for example mature microRNA sequencing, and therefore from now on we will use the term ‘RNA’ in the context of sequencing.

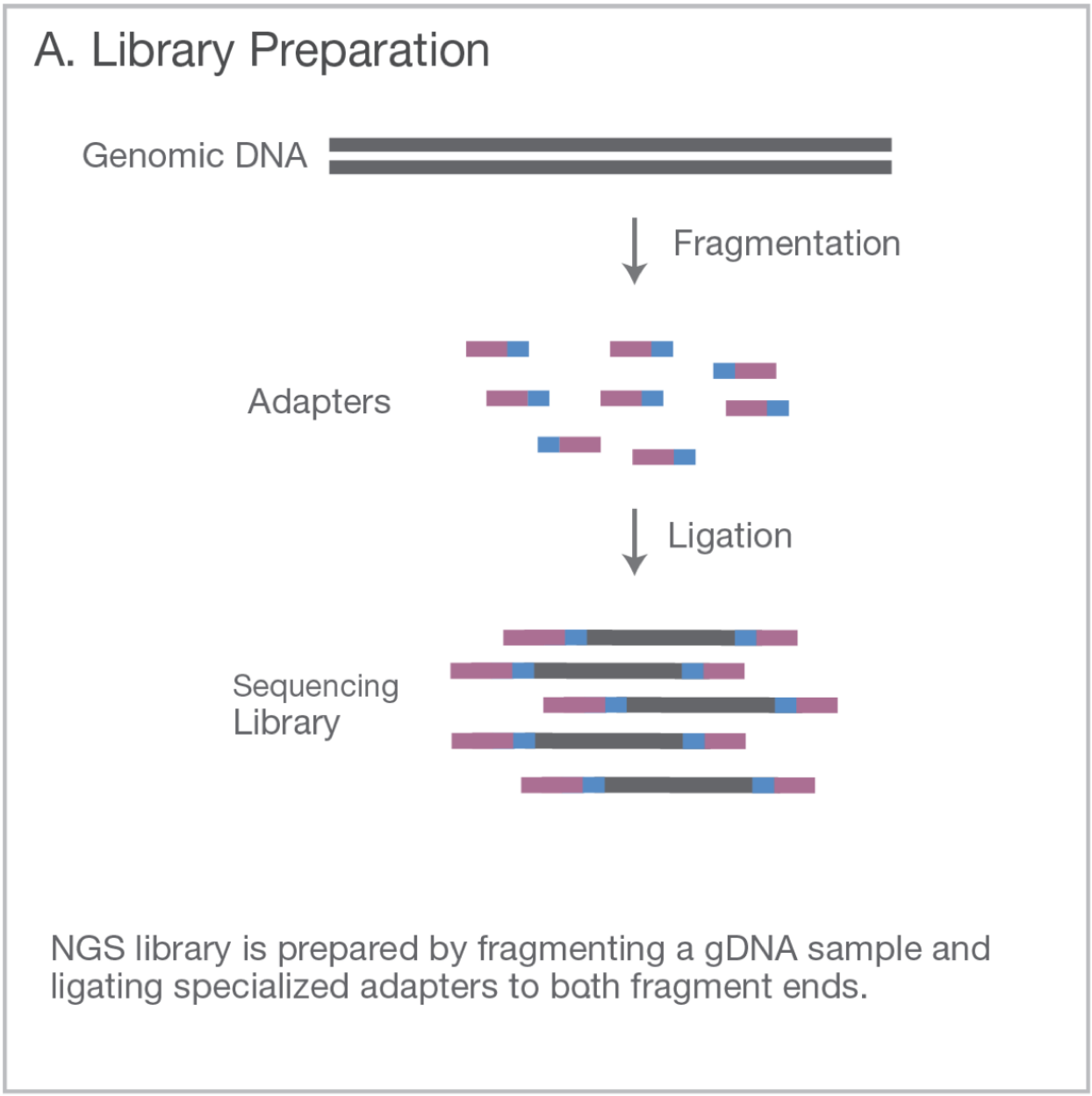
Note that RNA sequencing is basically a ‘snapshots’ in time, i.e. each dataset represents the expression of all available RNA in the sample at a particular time. In most cases, only one ‘snapshot’ is possible per sample; for example, one hippocampal tissue can be thoroughly studied via RNA sequencing, because the tissue is dissolved in the process.

Can the RNA be degraded in the process? Yes, but if the RNA isolation is done quickly and using proper protocols then most degradation can be avoided. The degradation can either be

from external sources -- our hands contain Ribonuclease (RNase), or from internal sources -- the cells start to secrete RNases the moment cell death starts. RNA from fixed tissues is also partly degraded. This will happen in formalin-fixed tissues, and even more severely in formalin-fixed and paraffin-embedded (FFPE) tissues.

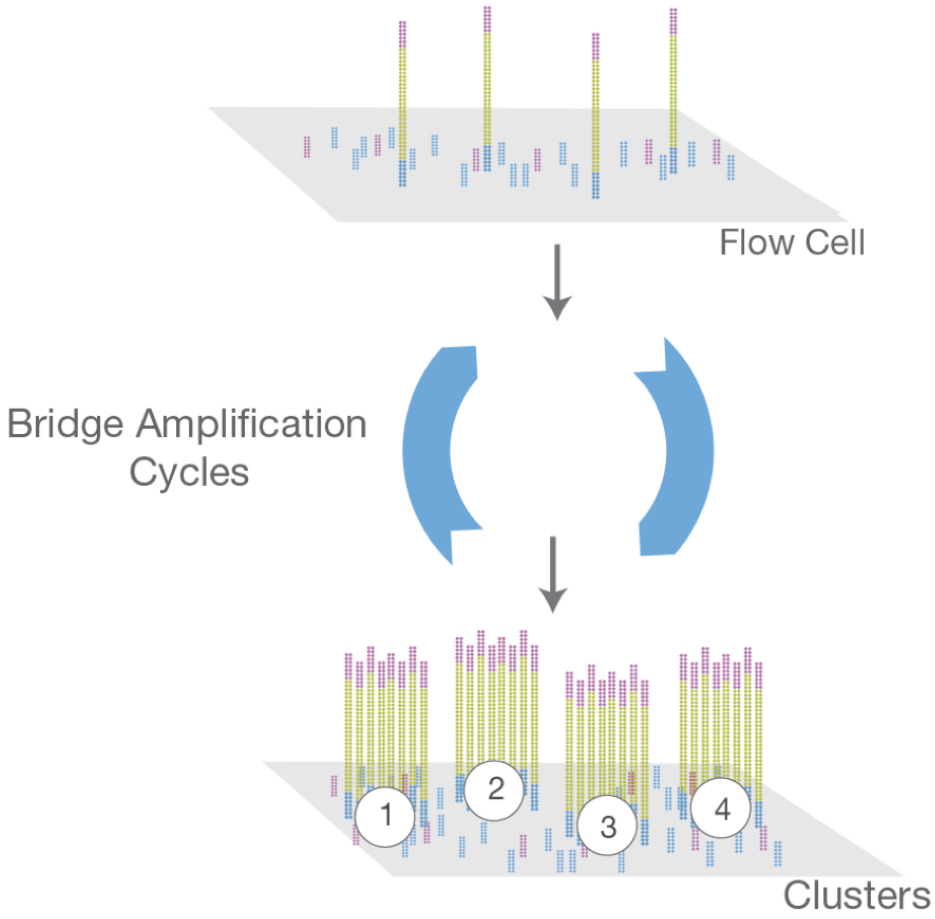
**Step 2: Amplification and sequencing of RNA molecules**

Illumina sequencing (pictures from here)

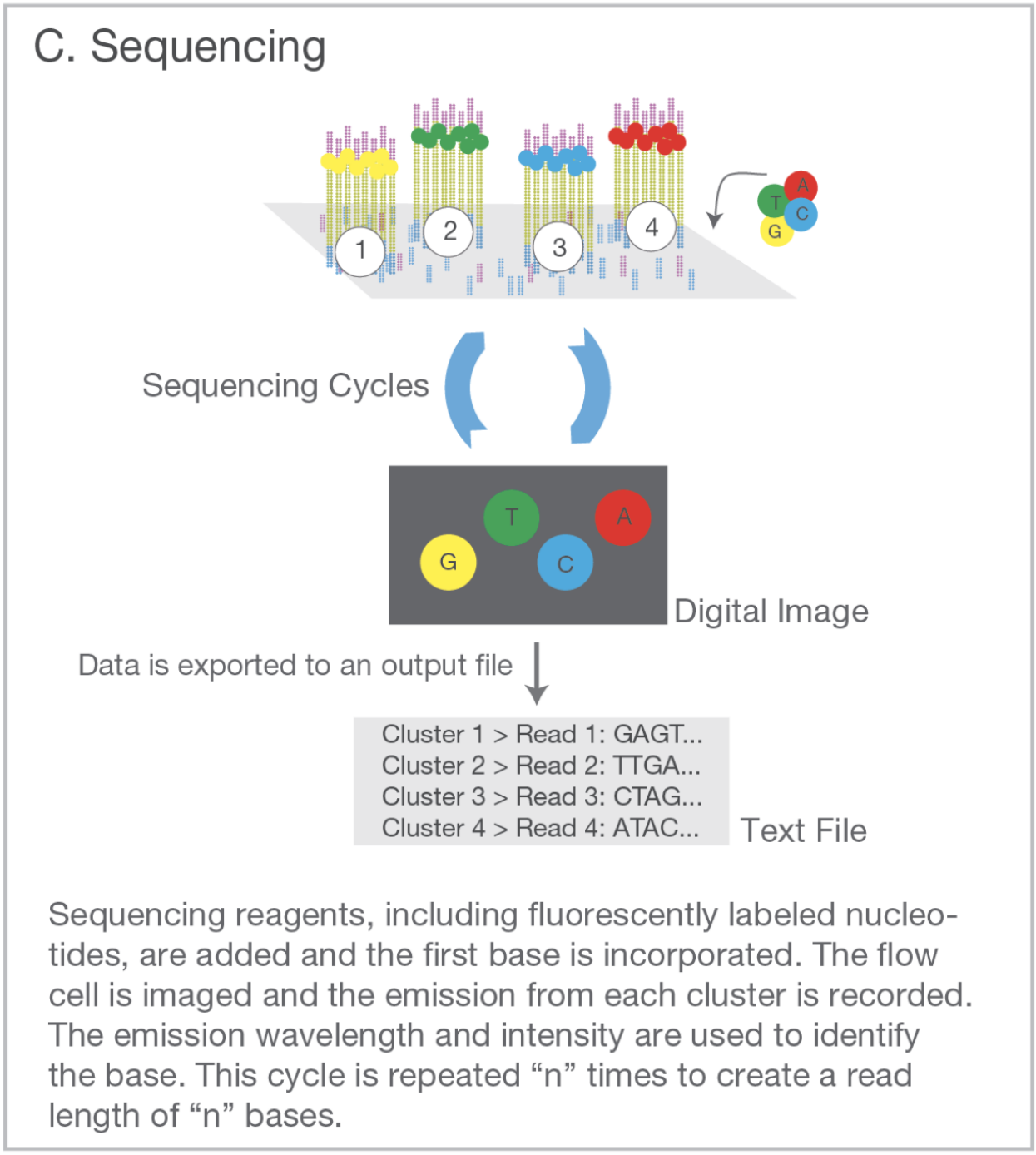



Note that the figure starts from double stranded genomic DNA, but the process is exactly the same for RNA as well, the only difference is that the starting point is double stranded cDNA.

## B. Cluster Amplification

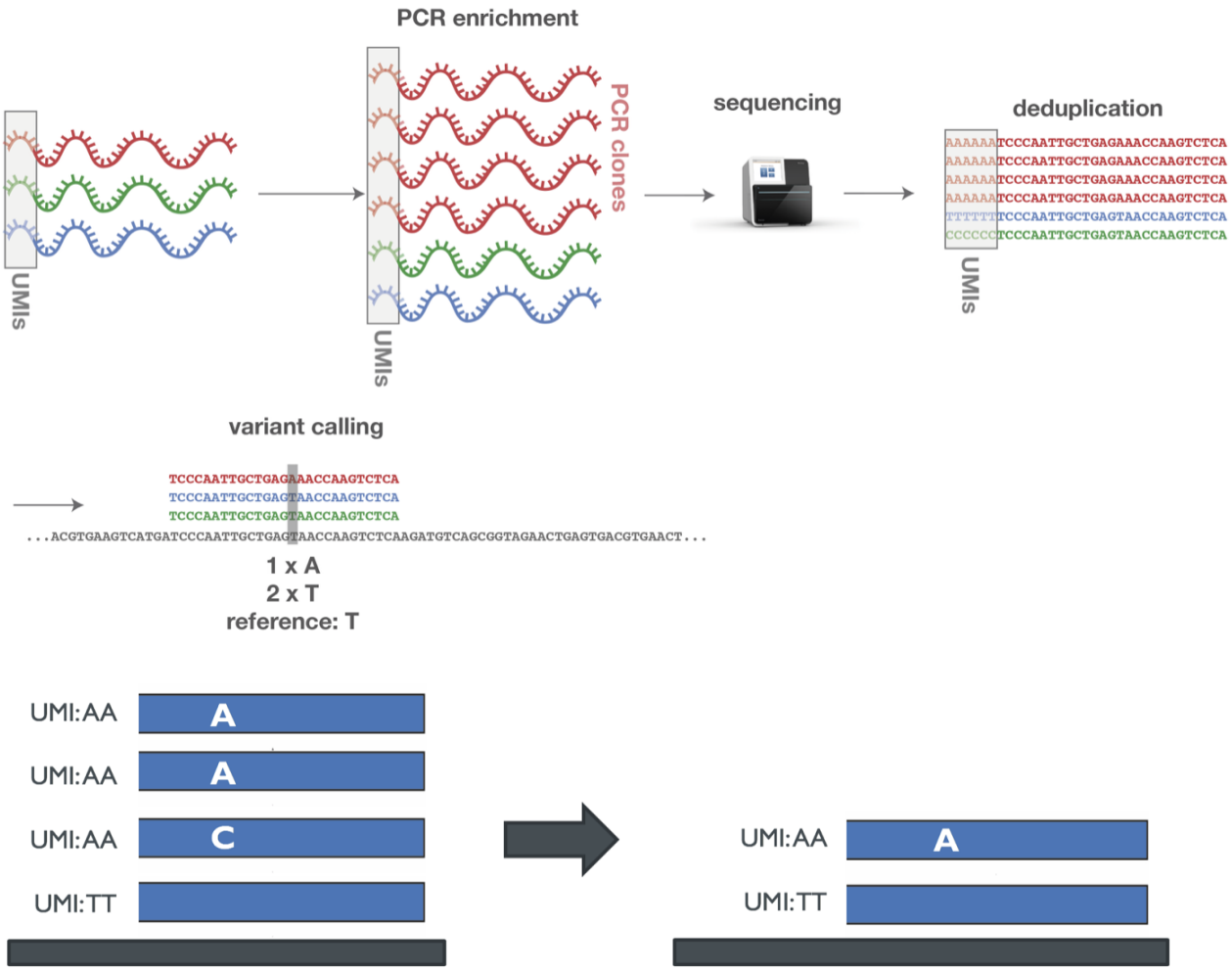



Library is loaded into a flow cell and the fragments are hybridized to the flow cell surface. Each bound fragment is amplified into a clonal cluster through bridge amplification.



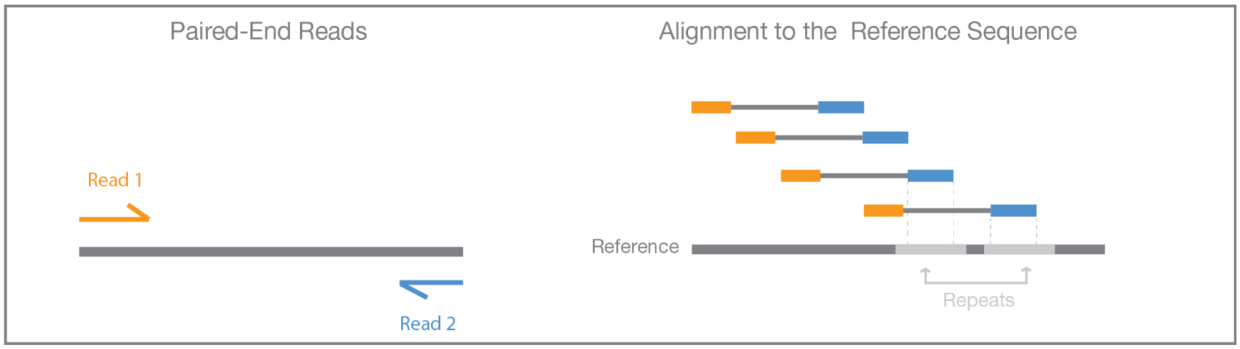

Read about the problem with PCR/Bridge Amplification in exercise 1. The main problem is that the amplification is not linear.

Unique molecular identifiers (UMI, pictures from here)
One solution for the non-linear amplification problem discussed above is to add a short unique sequence for each cDNA molecule, before the amplification.




<u>Paired-end sequencing (picture from here)</u>

The idea is to sequence from both sides of the amplicon. This helps to accurately align the reads to the genome or transcriptome (i.e. the mRNA sequence, a.k.a the genes).

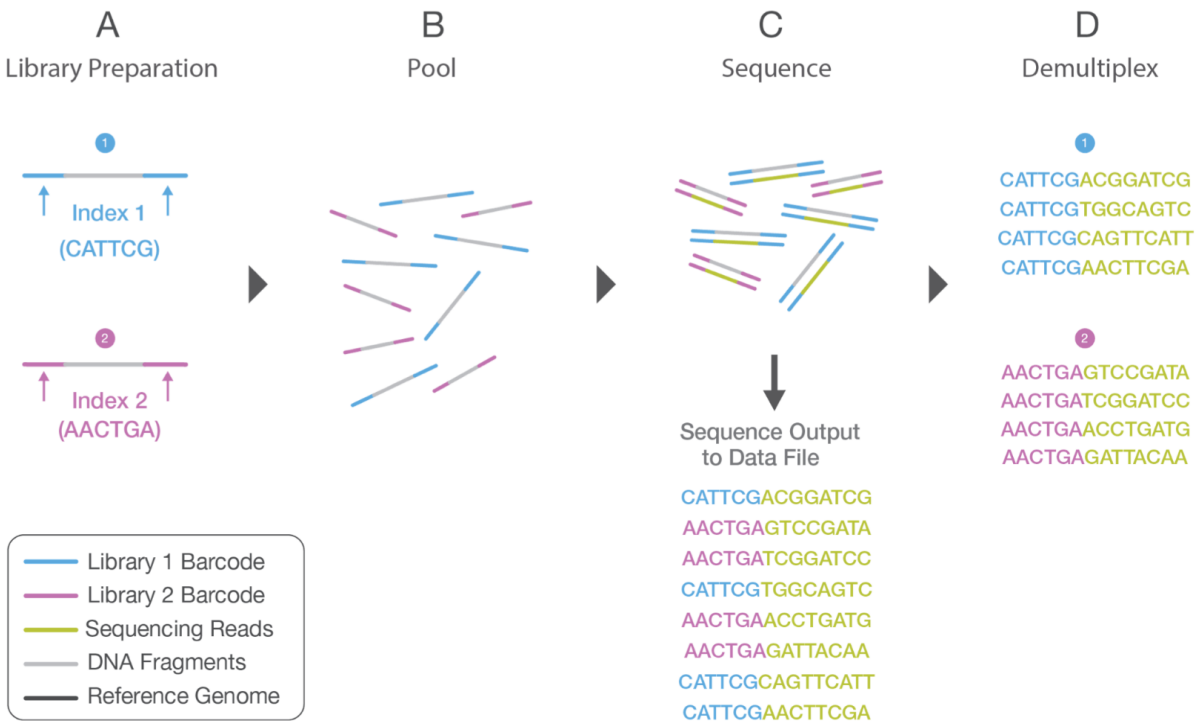



Question: How long should the reads be?

Answer: The longer the better, but the total number of bases generated in a sequencing machine is limited.

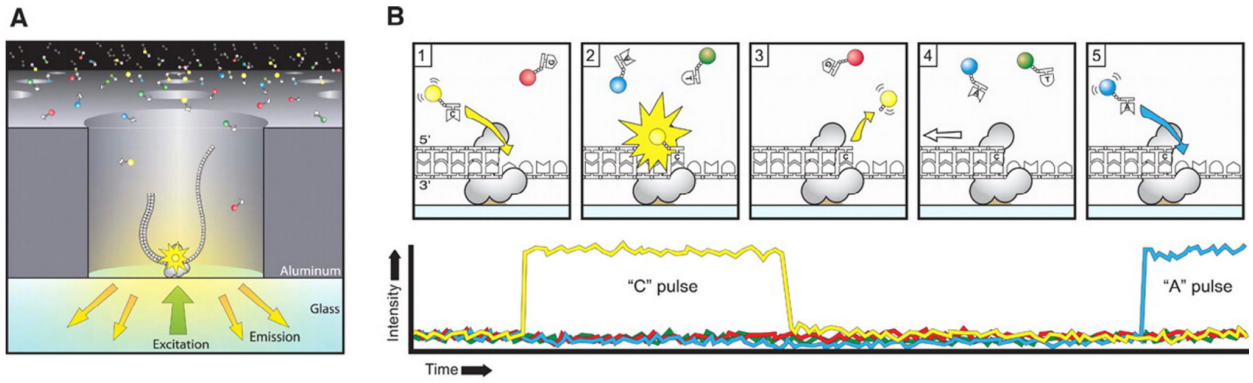
Multiplexing (picture from here)

The idea is to sequence several conditions together, using the same sequencing run of the machine. This is done by adding a unique barcode for each condition. The barcode is read during the sequencing, and reveals the condition identity.

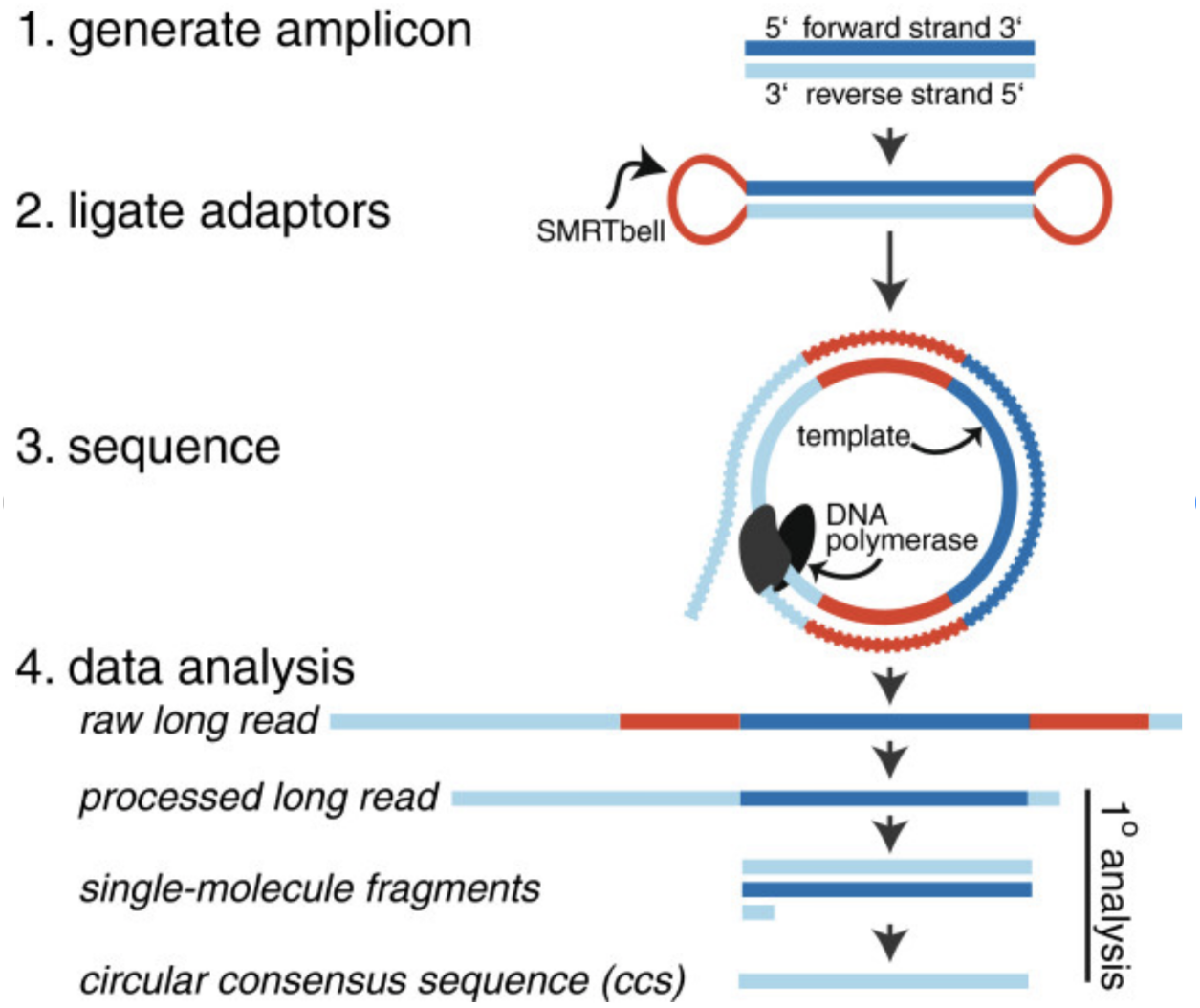



PacBio (pictures from here and here)

This is the leading alternative to Illumina. Originally this technology was single molecule and in real-time. The promise was that we can do single molecule sequencing (no amplification), that gives full length mRNA (no RNA fragmentation as in Illumina), and in real time (i.e. much faster than Illumina). The idea is to place each single DNA molecule with one polymerase in a small comportment (zero-mode waveguide) and then to excite and read information from these comportments.




The problem was that this technology gave many sequencing errors, which reduced the quality of the sequencing. The solution now is to avoid the real time part, and to average many sequencing runs of the same molecule. This method is now capable of sequencing full mRNA without amplification.



Phred score and FASTQ format (picture from here)
The quality score of the base is called the Phred score, and gives a measure of how accurate was the sequencing in that base.
Formally:
Q = -log10(P-value)
However, this is not really the P-value of a sequencing error.
In practice, for Illumina 1<=Q<=40, Q<15 is considered very bad quality and Q>30 is considered ok.
Each Q value is represented by one ASCII letter. In the figure below, L represents Illumina encoding, and P represents PacBio encoding.

```
LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL.....................................................
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

```
0.2.....................26...31........41
0..................20........30........40........50.....................................93
```

FASTQ format (picture from here)
This is the output of the sequencing machine, and it is considered the 'raw' sequencing data.
For each read, there are two equal parts. The first is the actual sequence, and the second is the quality score of each base.
The format is:
@ read title
SEQUENCE
+ read title
QUALITY
Here is a real example:

```
@SRR001666.1 071112_SLXA-EAS1_s_7:5:1:817:345 length=36
GGGTGATGGCCGCTGCCGATGGCGTCAAATCCCACC
+SRR001666.1 071112_SLXA-EAS1_s_7:5:1:817:345 length=36
IIIIIIIIIIIIIIIIIIIIIIIIIIII9IG9IC
```

Question: when should we use Python and when should we use Matlab/R?
Answer: Sequence analysis and analysis of large datasets are better in Python. Statistical analysis, matrix analysis, image analysis and plotting are better in Matlab/R

Regular Expressions in Python
As a motivation for this topic, say that we have a Fastq file with our data, and we want to check that the adapters were properly removed. The same goes for barcodes and UMIs. How can we remove them from the sequencing data? Regular expressions can do that, and they can also do much more.
In Linux, there are many built-in commands that can process sequencing reads (or any other sequence of letters). However, using regex makes it so that we can process sequencing reads

in Python using any system, including windows. Moreover, regex is almost the same in all programming languages, so even Python is not necessary.

Below is a quick explanation about regular expressions, some of it is taken from here, and here.
Regular expressions (or 'regex') are essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module.
You can ask questions such as "Does this string match the pattern?", or "Is there a match for the pattern anywhere in this string?". You can also use REs to modify a string or to split it apart in various ways.

Quick example

```
>>> import re
>>> m = re.search('[a-z]+', 'tempo')
>>> m
<re.Match object; span=(0, 5), match='tempo'>
```

The search pattern is [a-z]+, which means search for all letters from a to z.
The square brackets mean 'or', for example [aTg] means a or T or g.
+ means length of 1 and above for the match.
Now we can query the match object for information about the matching string.

```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

In addition to the search() method, there are additional methods:

| Method/Attribute | Purpose |
|---|---|
| `match()` | Determine if the RE matches at the beginning of the string. |
| `search()` | Scan through a string, looking for any location where this RE matches. |
| `findall()` | Find all substrings where the RE matches, and return them as a list. |

Special characters:

```
^ Matches the start of the string
$ Matches the end of the string
* Causes the resulting RE to match 0 or more repetitions
+ Causes the resulting RE to match 1 or more repetitions
. this matches any character except a newline
? Causes the resulting RE to match 0 or 1 repetitions
*?, +?, ?? The '*', '+', and '?' qualifiers are all greedy; they match
as much text as possible. Adding ? after the qualifier makes it
perform the match in non-greedy or minimal fashion; as few characters
as possible will be matched.
{m} Specifies that exactly m copies of the previous RE should be
matched
{m,n} Causes the resulting RE to match from m to n repetitions
| the choice operator (‘a|b’ means either a or b)
```

```
\ Either escapes special characters (permitting you to match
characters like '*', '?', and so forth), or signals a special
sequence
\w means a word character [a-zA-Z0-9_]
\W anything but word character
\n end of line
\t tab
```

```
[] Used to indicate a set of characters. In a set: Characters can be
listed individually, e.g. [amk] will match 'a', 'm', or 'k'. Ranges
of characters can be indicated by giving two characters and
separating them by a '-', for example [a-z] will match any lowercase
letter, [0-5][0-9] will match all the two-digits numbers from 00 to
59. Special characters lose their special meaning inside sets. For
example, [(+*)] will match any of the literal characters '(', '+',
'*', or ')'. If the first character of the set is '^', all the
characters that are not in the set will be matched.
```

Examples (from here):

- `.at` matches any three-character string ending with "at", including "hat", "cat", "bat", "4at", "#at" and " at" (starting with a space).
- `[hc]at` matches "hat" and "cat".

- `[^b]at` matches all strings matched by `.at` except "bat".
- `[^hc]at` matches all strings matched by `.at` other than "hat" and "cat".
- `^[hc]at` matches "hat" and "cat", but only at the beginning of the string or line.
- `[hc]at$` matches "hat" and "cat", but only at the end of the string or line.
- `\[.\]` matches any single character surrounded by "[" and "]" since the brackets are escaped, for example: "[a]", "[b]", "[7]", "[@]", "[]]", and "[ ]" (bracket space bracket).
- `s.*` matches s followed by zero or more characters, for example: "s", "saw", "seed", "s3w96.7", and "s6#h%(>>>m n mQ".
- [hc]?at matches "at", "hat", and "cat".
- [hc]*at matches "at", "hat", "cat", "hhat", "chat", "hcat", "cchchat", and so on.
- [hc]+at matches "hat", "cat", "hhat", "chat", "hcat", "cchchat", and so on, but not "at".
- cat|dog matches "cat" or "dog".

Lookahead and lookbackwards

`(?=...)`

Matches if `...` matches next, but doesn't consume any of the string. This is called a *lookahead assertion*. For example, `Isaac (?=Asimov)` will match `'Isaac '` only if it's followed by `'Asimov'`.

`(?!...)`

Matches if `...` doesn't match next. This is a *negative lookahead assertion*. For example, `Isaac(?!Asimov)` will match `'Isaac '` only if it's *not* followed by `'Asimov'`.

`(?<=...)`

Matches if the current position in the string is preceded by a match for `...` that ends at the current position. This is called a *positive lookbehind assertion*. For example:

```
>>> m = re.search('(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
>>> m = re.search('(?<=-)\w+', 'spam-egg ok-computer')
>>> m
<re.Match object; span=(5, 8), match='egg'>
```

```
>>> m = re.findall('(?<=-)\w+', 'spam-egg ok-computer')
>>> m
['egg', 'computer']
```

`(?<!...)`

> Matches if the current position in the string is not preceded by a match for `...`. This is called a *negative lookbehind assertion*.

**Step 3: Alignment of RNA sequences against the genes/genome**

Softwares for sequence alignment

A commonly used aligner is Bowtie2. Bowtie2 is a fast aligner of short Illumina sequences into the genome. Bowtie2 has two modes:

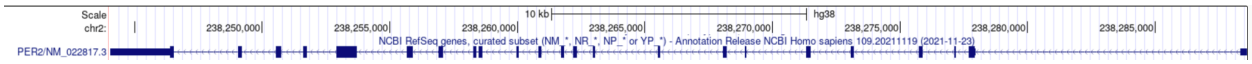
a) Bowtie2 global alignment mode ('end-to-end') - this mode is not recommended! (why?)

```
Read:       GACTGGGCGATCTCGACTTCG
Reference:  GACTGCGATCTCGACATCG


Alignment:
  Read:       GACTGGGCGATCTCGACTTCG
              |||||  |||||||||| |||
  Reference:  GACTG--CGATCTCGACATCG
```

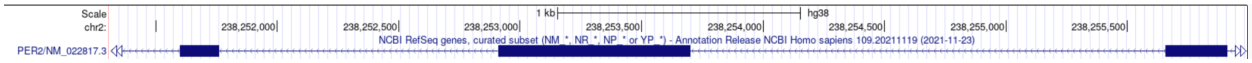
b) Bowtie2 local alignment (only use this mode)

```
Read:       ACGGTTGCGTTAATCCGCCACG
Reference:  TAACTTGCGTTAAATCCGCCTGG

Alignment:
   Read:       ACGGTTGCGTTAA-TCCGCCACG
                  ||||||||| ||||||
   Reference:  TAACTTGCGTTAAATCCGCCTGG
```

Bowtie2 can not handle large gaps in the alignment, which can be a result of introns.
Let's use ucsc genome browser to look at one gene (per2), and see his exons and introns:




Zoom-in:




If the read comes completely from an exon, then Bowtie2 will work fine. But if a part of the read is in exon1, and some is in exon2, then Bowtie2 will fail.
An aligner such as HISAT2 will do a better job at aligning in such cases, and therefore is considered a more general aligner.

Question: where should we expect failures in the alignment?
Answer: at the beginning and at the end of the read, and also near exons/introns junctions in the alignment.

Question: can we align against the genes instead of aligning to the genome? The reads should come from mature RNA and therefore no introns are expected, right? If we are aligning against the genes, then Bowtie2 should be perfect!
Answer: It is possible to align against the genes, instead of the genome. In most animals, we don't have the genome so we don't really have a choice. However, if we do have a genome, aligning against it will produce better results. One reason is because the RNA sequences actually include, in many cases, introns. We sequence mature RNA (i.e. from outside the nucleus), together with precursor RNA (i.e. from the nucleus). The second reason is that we have extensive alternative splicing for the genes, and therefore the genes' sequences are not complete.

Mapping quality

Aligners characterize their degree of confidence by reporting a mapping quality:
MAPQ = -10 log10 p, where p is an estimate of the probability that the alignment does not correspond to the read's true point of origin. Mapping quality is sometimes abbreviated MAPQ. Mapping quality is related to "uniqueness". An alignment is unique if it has a much higher alignment score than all the other possible alignments. These cases should be with MAPQ ~40.

SAM/BAM format

A compact way to pack all the information about the alignment into a text file.
Say that we have these following alignments:

```
Coor      12345678901234  5678901234567890123456789012345
ref       AGCATGTTAGATAA**GATAGCTGTGCTAGTAGGCAGTCAGCGCCAT

+r001/1        TTAGATAAAGGATA*CTG
+r002         aaaAGATAA*GGATA
+r003       gcctaAGCTAA
+r004                     ATAGCT..............TCAGC
-r003                           ttagctTAGGC
-r001/2                                        CAGCGGCAT
```

In SAM format then converts the alignment into this:

```
r001   99 ref  7 30 8M2I4M1D3M = 37  39 TTAGATAAAGGATACTG *
r002    0 ref  9 30 3S6M1P1I4M *  0   0 AAAAGATAAGGATA    *
r003    0 ref  9 30 5S6M       *  0   0 GCCTAAGCTAA       * SA:Z:ref,29,-,6H5M,17,0;
r004    0 ref 16 30 6M14N5M    *  0   0 ATAGCTTCAGC       *
r003 2064 ref 29 17 6H5M       *  0   0 TAGGC             * SA:Z:ref,9,+,5S6M,30,1;
r001  147 ref 37 30 9M         =  7 -39 CAGCGGCAT         * NM:i:1
```

The SAM format includes all the original information, including the gaps in the alignments, and the locations of mismatches.

Question: how can we move from sequences aligned to the genome to the number of sequences per gene (the vector that we wanted)?
Answer: option (1) is to just count… we can count the number of unique alignments for each gene. Option (2) is to use software, such as Cufflinks, which can account for non-unique alignments as well. Cufflinks can also detect differential expression of genes, which will be the topic of the next lesson.

Question: do we really need to align the RNAseq reads if we only care about the quantification of the genes?
Answer: probably not… the idea of pseudo-alignment can be used to rapidly determine the compatibility of reads with targets, without the need for full alignment. The result is the same - quantifying the abundances of transcripts, but much faster.

Two tools do that, and they are basically the same:
Salmon&Kallisto

Now that we finally have the vector that contains the expression of all the genes, we need to normalize it. Why do we need normalization and how can it be done? Read about it in exercise 1.