Variants calling from DNA-seq data

CE7412: Computational and Systems Biology

2024-06-06

Contents

1 Sequencing

DNA/RNA sequencing is the determination of the order of the four nucleotides in a nucleic acid molecule.

1.1 First-generation sequencing

The emergence of first-generation sequencing emerged in 1970s when Allan Maxam and Walter Gilbert developed a chemical method for sequencing, followed by Frederick Sanger who developed the chain-terminator method. Sanger method became the more commenly used first-generation sequencing method. The steps in Sanger sequencing are similar to that of PCR, including denaturation, primer anealing, and complementary strand synthesis. However, sample DNA is divided into for reaction tubes labelled by di-deoxynucleotide tripospates: ddATP, ddGTP, ddCTP, and ddTTP. The synthesis terminates in DNA fragments of varying length and the fragments are then separated by size using gel electophoresis.The DNA bands are then graphed by autoradiography and the order of the nucleotide bases on the DNA sequence can be directly read from Xray film.

1.2 Next-generation sequencing (NGS)

Next-generation sequencing (NGS) is a massively parallel sequencing technology that offers ultra-high throughput, scalability, and speed. Compared to earlier sequencing technology such as Sanger sequencing which focuses on one fragment at a time, NGS provides sequencing information on multiple fragments simultaneously, producing counts data.

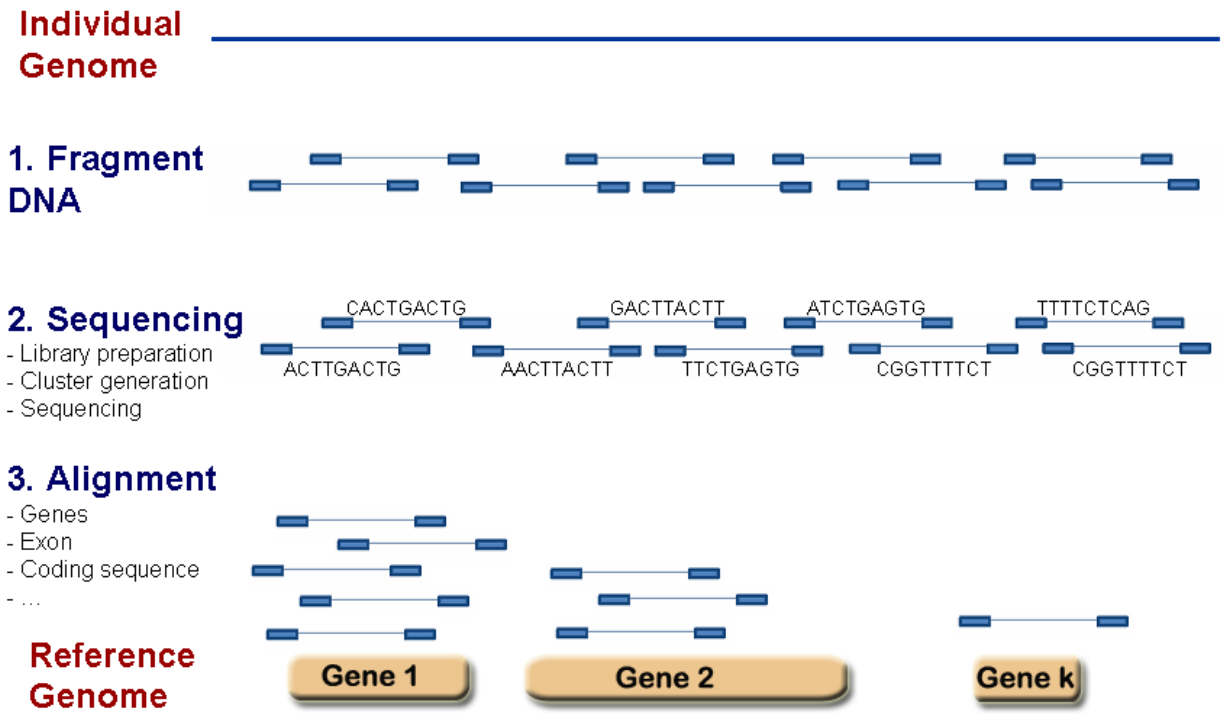


Figure 1: Next-generation sequencing workflow

Next-Generation Sequencing refers to a class of technologies that sequence millions of short DNA fragments in parallel, with a relatively low cost. The length and number of the reads differ based on the technology. Currently, there are three major commercially available platforms/technologies for NGS: (1) Illumina, (2)

Roche, and (3) Life Technologies. The underlying chemistry and technique used in each platform is unique and affects the output.

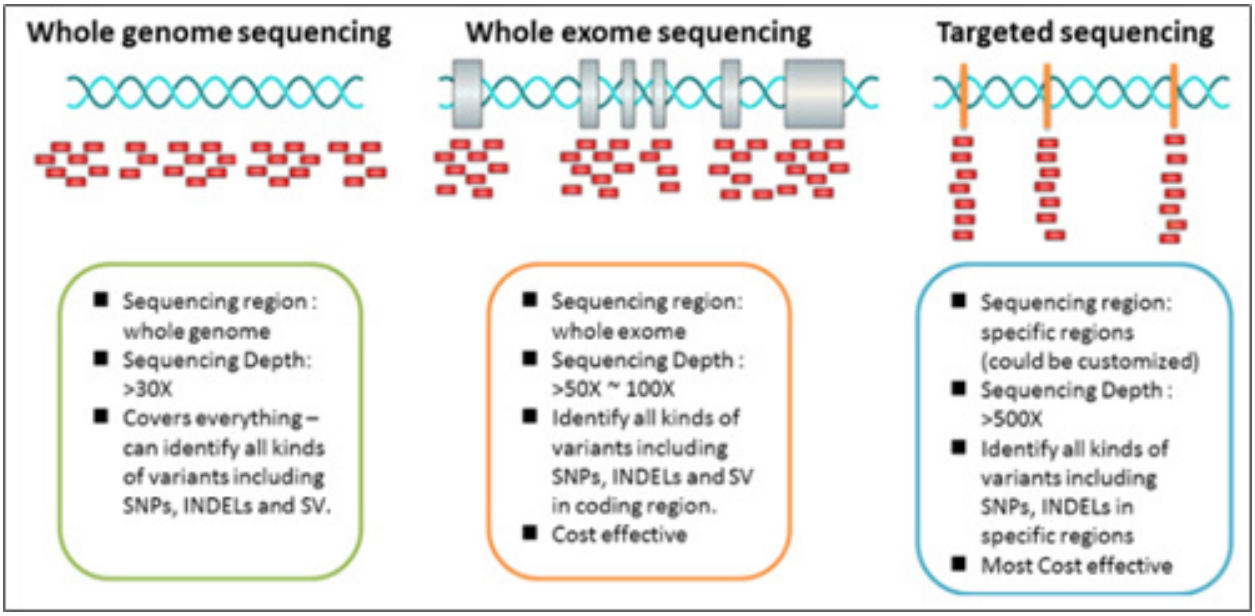
NGS sequencing enables a wide variety of applications, allowing researchers to ask virtually any question related to the genome, transcriptome, or epigenome of any organism. NGS methods differ primarily by how the DNA or RNA samples are prepared and the data analysis options used. The technology is used to determine the order of nucleotides in entire genomes or targeted regions of DNA or RNA.

1.3 Third-generation sequencing

Third generation sequencing (TGS) is recent sequencing technique that addresses drawbacks of NGS, such as the needs for long reads and poor resolution. The foundation of TGS emerged when DNA polymerase was used to obtain a sequence o single DNA molecules, which involves (i) direct imaging of individual DNA molecules using advanced microscopy techniques and (ii) nanopore sequencing technique in which a single molecule of DNA is threaded through a nanopore.

In general, there are two TGS technologies available: (i) Pacific Bioscience (PacBio) single molecule real time sequencing, and (ii) Oxford nanopore technologies.

2 Whole-genome sequencing (WGS) vs Whole-exome sequencing (WES)



2.1 WGS (Whole-genome sequencing)

WGS is a comprehensive method of analyzing the entire genomic DNA of a cell at a single time by using sequencing techniques such as Sanger sequencing, shotgun approach, or high throughput NGS sequencing. It is also known as full genome sequencing or complete genome sequencing. Whole-genome sequencing entails the sequencing all of an organism's chromosomal DNA as well as DNA in mitochondria, and for plants, the DNA in the chloroplast at a single time. WGS enables scientists to read the exact sequence of all the letters that make up the complete set of DNA.

2.2 WES (Whole-exome sequencing)

WES focuses on the genomic proteincoding regions (exons). Although WES requires additional reagents (probes) and some additional steps (hybridization), it is a cost-effective, widely used NGS method that requires fewer sequencing reagents and takes less time to perform bioinformatic analysis compared to WGS. Although the human exome represents only 1-5% of the genome, it contains approximately 85% of known disease-related variants.1 As such, researchers performing WES achieve comprehensive coverage of coding variants such as single nucleotide variants (SNVs) and insertions/deletions (indels). Despite lengthier sample preparation due to the additional target enrichment step, scientists benefit from quicker sequencing and data analysis compared to WGS. WES provides greater sequencing depth for researchers interested in identifying genetic variants for numerous applications, including population genetics, genetic disease research, and cancer studies.

3 Sequencing depth and read quality

3.1 Sequencing depth and coverage

The biological results and interpretation of sequencing data for different sequencing applications are greatly affected by the number of sequenced reads that cover the genomic regions.

The sequencing *depth* measures the average read abundance and it is calculated as the number of bases of all sequenced short reads that match a genome divided by the length of that genome.

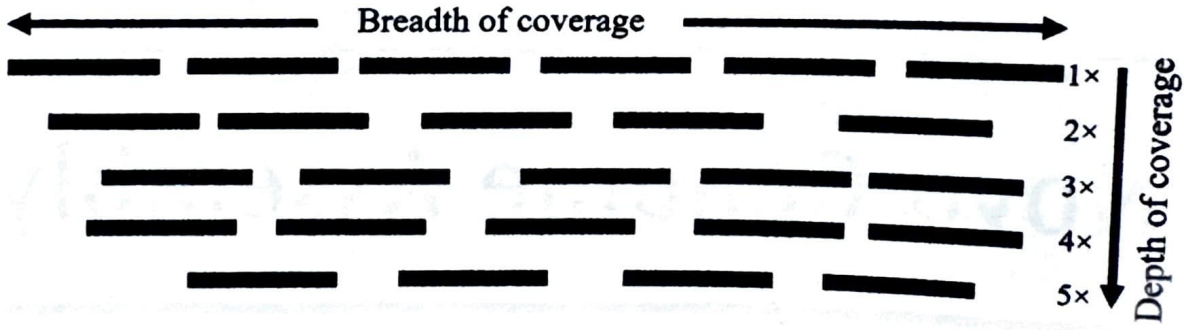


Figure 2: Sequence coverage and depth

The *coverage* gives the density of reads. If the reads are equal

$$\text{Coverage} = \frac{\text{read length}(bp) \times \text{number of reads}}{\text{genome size}(bp)}$$

If the reads are not equal in length, the coverage is calculated as

$$\text{Coverage} = \frac{\sum_{i=1}^{n} \text{lenght of read } i}{\text{genome size}(bp)}$$

where n is the number of sequece reads.

3.2 Base call quality

The process of inferring the base at a specific position of the sequenced DNA fragment during the sequencing process is called the *base calling*. Most sequencing platforms are equipped with base calling programs that

assign a *Phred* quality score to measure the accuracy of each base called. The **Phred quality score** (Q-score) transforms the probability of calling a base wrongly into an integer core that is easy to interpret. The Phred score is defined as

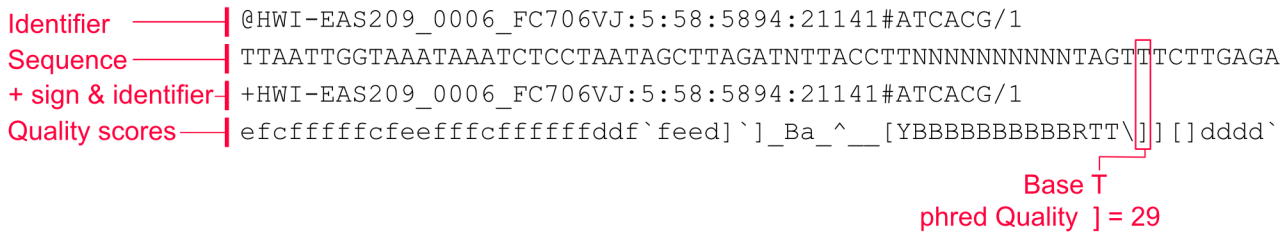
$$p = 10^{-Q/10}$$

$$Q = -10 \log_{10}(p)$$

where p is the probability of the base call being wrong.

4 Fasta and Fastq formats

High-throughput sequencing reads are usually output from sequencing facilities as text files in a format called "FASTQ" or "fastq". This format depends on an earlier format called FASTA. The FASTA format was developed as a text-based format to represent nucleotide or protein sequences. An extension of the FASTA format is FASTQ format. This format is designed to handle base quality metrics output from sequencing machines. In this format, both the sequence and quality scores are represented as single ASCII characters. The format uses four lines for each sequence, and these four lines are stacked on top of each other in text files output by sequencing workflows.



Credit: https://compgenomr.github.io/book/fasta-and-fastq-formats.html

A **Phred quality score (Q score)** is a metric used to measure the accuracy of a sequencing platform. It indicates the probability that a given base is called incorrectly by the sequencer. Q score is usually indicated by an ASCII character.

5 DNA-Seq analysis pipeline

The DNA-Seq analysis pipeline identifies somatic variants within whole exome sequencing (WXS) and whole genome sequencing (WGS) data. Somatic variants are identified by comparing allele frequencies in normal and tumor sample alignments, annotating each mutation, and aggregating mutations from multiple cases into one project file.

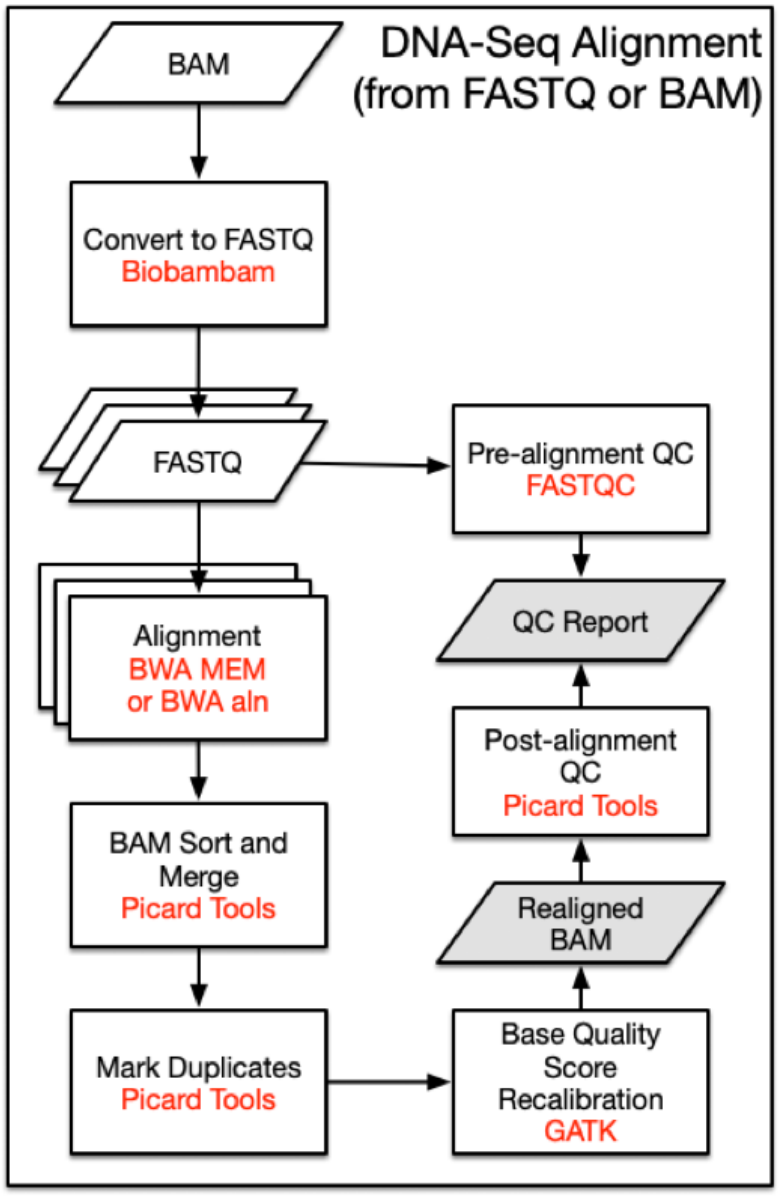
5.1 Steps in DNA-Seq analysis

Variant calling is the process by which we identify variants from sequence data:

- Base calling: Carry out whole genome or whole exome sequencing to create bases for each locations in FASTQ files.
- Filtering
- Read mapping: Align the sequences to a reference genome, creating BAM or CRAM files.
- Quality control

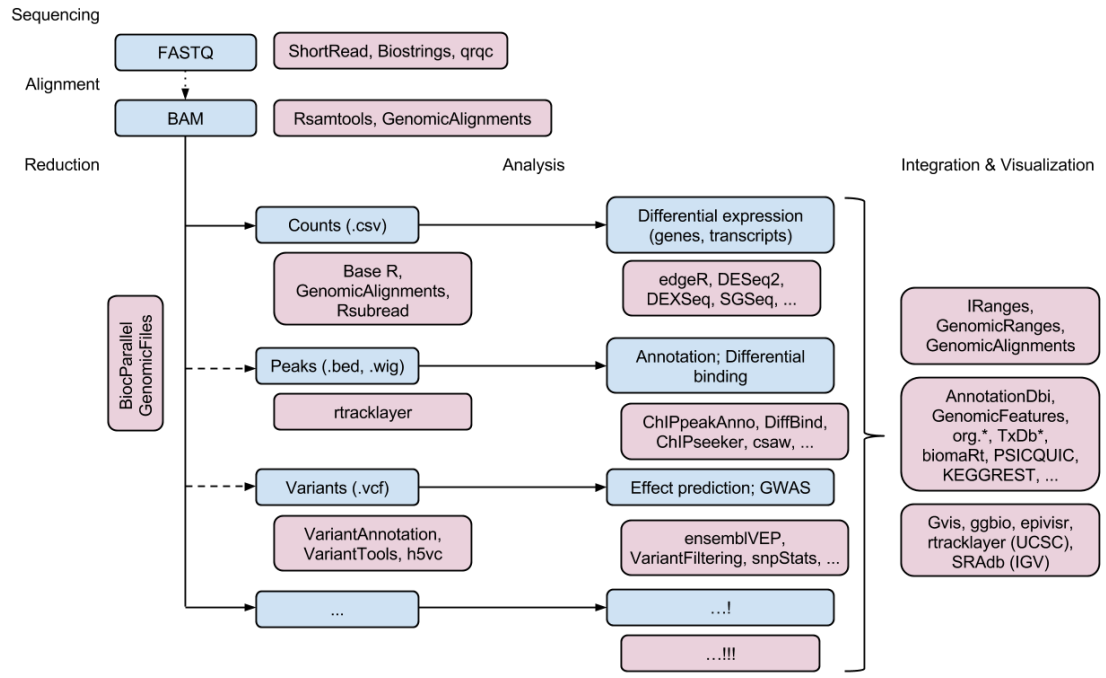
- Variant calling: Identify where the aligned reads differ from the reference genome and write to a VCF file.
- Filtering
- Population analysis: GWAS

5.2 Alignment pipeline (GDC)



Credit: https://docs.gdc.cancer.gov/Data/Bioinformatics_Pipelines/DNA_Seq_Variant_Calling_Pipeline/

5.3 Bioconductor resources for NGS analysis



Credit: https://dockflow.org/workflow/sequencing/

6 Read mapping

Read mapping refers to alignment of reads from high throughput analysis into a reference genome. The basic alignment algorithms do not work here because millions of reads have to be aligned to a reference genomes at the same time. Efficient *indexing algorithms* are needed to organize the sequence of the reference genome and the short reads in a memory efficient manner and to facilitate fast searching of the patterns. The commonly used data structures are:

- suffix trees
- suffix array
- Burrows-Wheeler transform (BWT)
- Full-text Minute space (FM-index)

6.1 Suffix Tree

Suffix tree is basically used for pattern matching and finding substrings in a given string of characters. It is constructed as a key and value pairs where all possible suffixes of the reference sequences as keys and the positions (indexes) in the sequence as the values. The following algorithm, known as Ukkonen's algorithm, constructs a suffix tree in a linear time.

For example, let's assume that our reference sequence consists of 10 bases as "CTTGGCTGGA$" where the positions are 0, 1, … 9, and $ is in the empty trailing position. Let's form the suffixes (keys) and indexes (values):

$ 10
A $ 9

GA $ 8
GGA $ 7
TGGA $ 6
CTGGA $ 5
GCTGGA $ 4
GGCTGGA $ 3
TGGCTGGA $ 2
TTGGCTGGA $ 1
CTTGGCTGGA $ 0

Note that each line consists of suffix (key) and a positon (value). Then a suffix tree can be made using key-value pairs as edges and nodes of the tree, respectively.

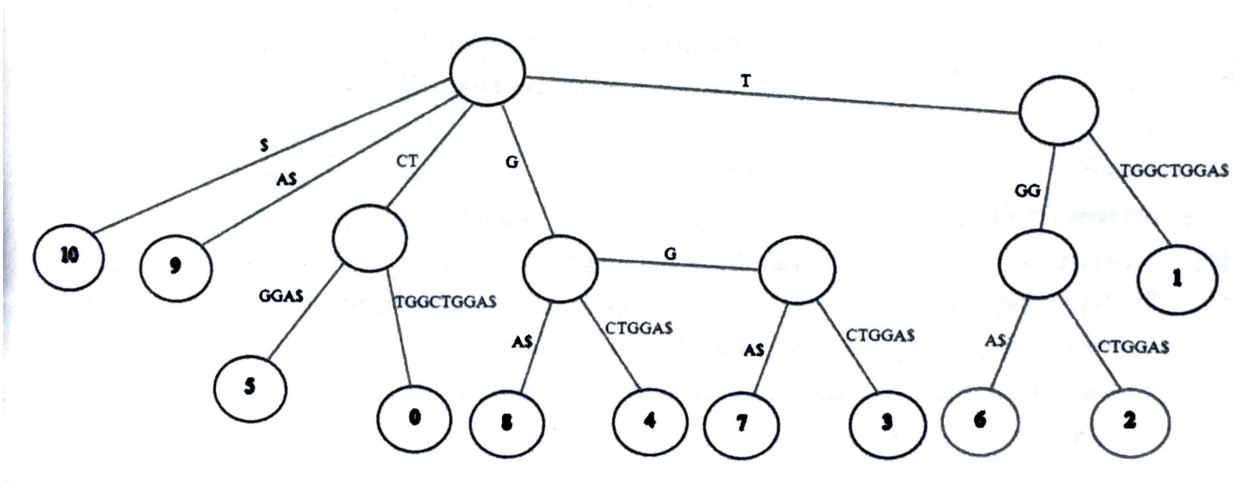


Figure 3: Suffix Tree

Once the suffix tree is build, there are several searching algorithms are available to find the location where a read maps. For instance, to find "TGG" in the tree, we will start looking for T and and then GG. And since there are two leaf nodes, "TGG" can map at positions 2 or 6.

6.2 Suffix Arrays

Suffix arrays can be constructed from suffix trees. It is basically a sorted array of all suffixes in a given sequence. We can quickly search for locations begin with suffixes "TGG" as 7 and 3.

6.3 Burrows-Wheeler Transform

The Borrows-Wheeler Transform (BWT) is a data structure that transform a string (sequence) into a compressible form that allows fast searching. The BWT is used by popular aligners like BWA and Bowtie. The BTW of the sequence 's' or $bwt(s)$ is computed by generating cyclic rotations of the sequences and then are sorted alphabetically to form a matrix called BWM (Borrows-Wheeler Matrix).

BWT could be obtained from a suffix array in a linear time complexity by using the following transform:

$$bwt(s)[i] = s[a[i] - 1] \text{ if } a[i] > 1 \text{ else } s[n]$$

i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10|
s | C | T | T | G | G | C | T | G | G | A |
a | 11| 10| 6 | 1 | 9 | 5 | 8 | 4 | 7 | 3 |

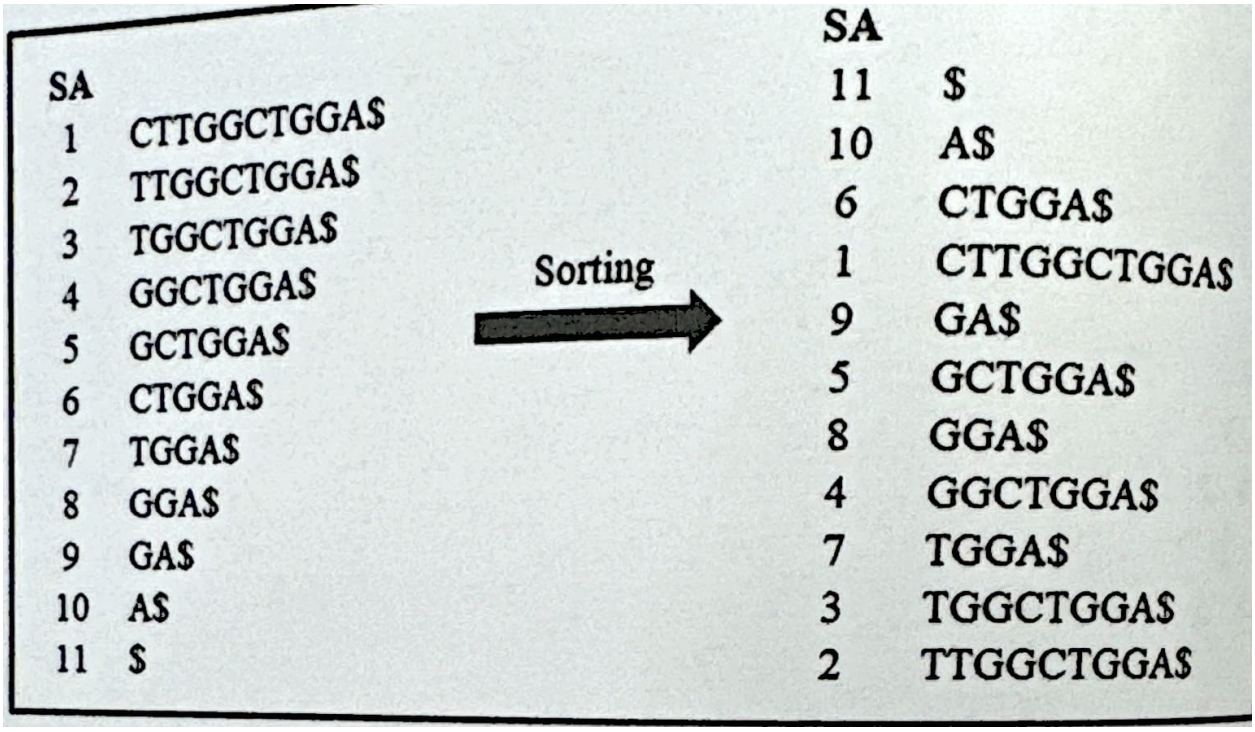


Figure 4: Suffix Array

|bwt| A | G | G | $ | G | G | T | T | C | T |

BWT serves two purposes. First, BWT columns are sorted alphabeticallly. Secondly, the second purpose is that it can find the matching position fast because it is reversible which property allows BWT to reverse to obtain BWM by a simple mapping.

7 De novo genome assembly

De novo genome assembly comes to play when there is no reference genome available for the organism. De novo genome assembly is a strategy to assemble a novel genome from scratch without the aid of a reference genome sequence. The de novo genome assembly aims to join reads into a contiguous sequence called a contig. Multiple contigs are joined together to form a scaffold and multiple scaffolds can be linked to form a chromosome.

Both single-end and paired-end reads can be used in the de novo assembly but paired reads are prefreed because they provide high quality reads. Assemblying the entire genome is usually challenging but with deep sequencing (with high coverage and depth), most of the challenges can be overcome.

The algorithms used for de novo genome assembly are:

- greedy approach
- overlap-lay-out consensus with Hamiltonian path
- de Bruijn graph with Eulerian path

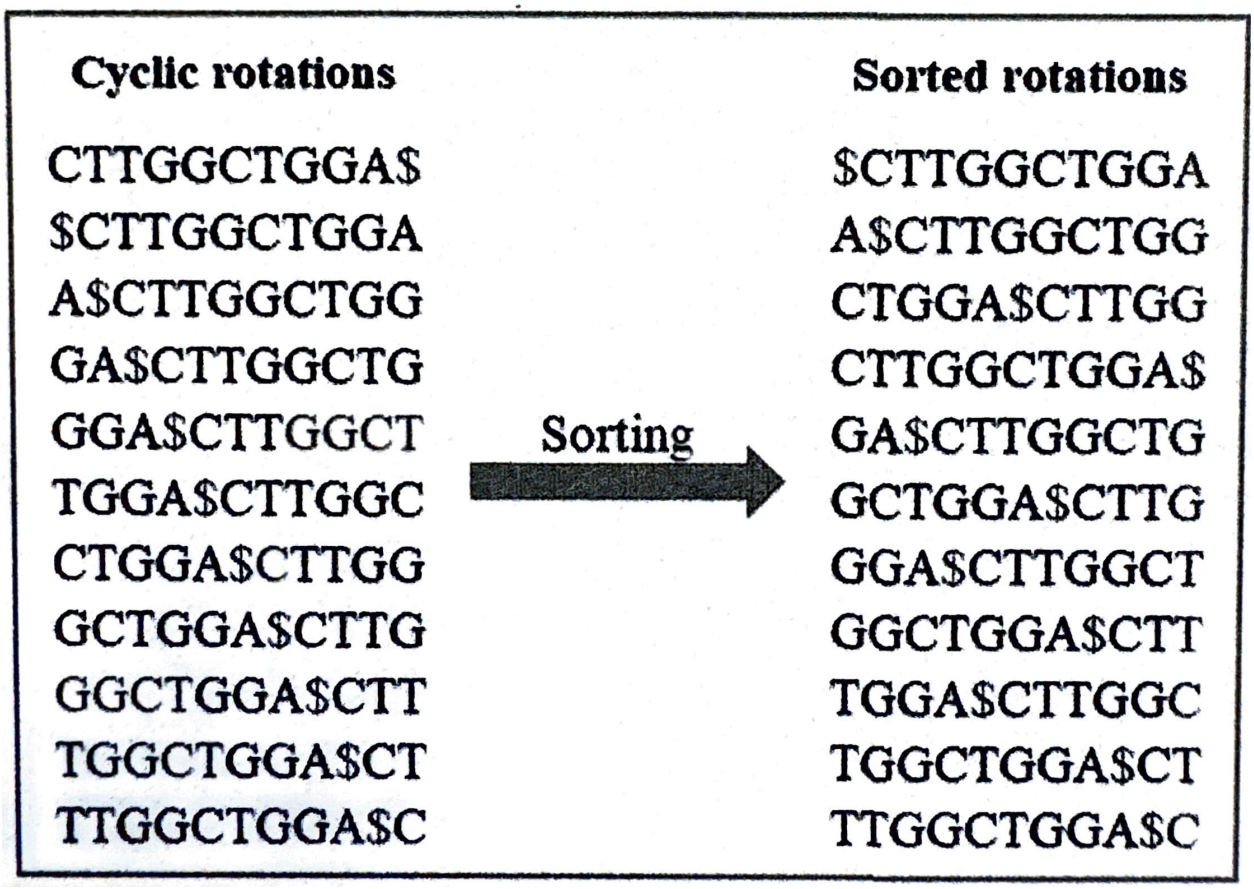


Figure 5: Burrows-Wheeler Transform

7.1 Greedy algorithm

It depends on similarity between reads to create a pile up of aligned sequences, which are collapsed to create contigs from the consensus sequences. The greedy algorithm uses pairwise alignment to compare all reads to identify the most similar reads with sufficient overlaps to be merged. The process continues repeatedly until there is no more merging.

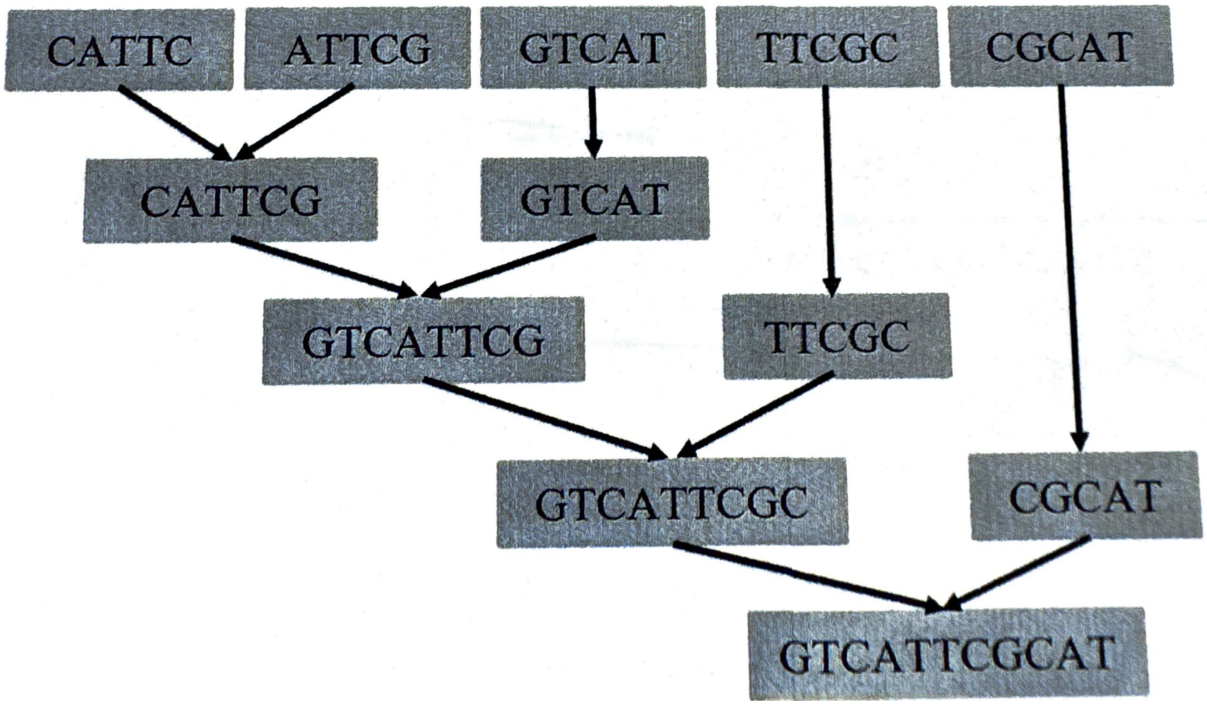


Figure 6: Greedy algorithm for de novo genome assembly

7.2 Overlap consensus graphs

The overlap-consensus algorithm performs pairwise alignment and then it represents the reads and the overlaps between the reads with graphs, where contiguous reads are the nodes and their overlaps are the edges. Each node r_i corresponds to a read and any two reads are connected by an edge $e(r_1, r_2)$ where the suffix is the first read maches the prefix of the second read. The algorithm then find the Hamiltonian parth of the graph that includes the nodes (reads) exactly once. Contigs are then created from the consensus sequencees of the overlapped suffixes and prefixes.

7.3 De Bruijn Graphs

In de Bruijn graphs, each read is broken into overlapping substrings of length k called overlapping k-mers. The k-mers are then represented by graphs in which each k-mer is a vertex. Any two nodes of any two k-mers that share a commeon prefix and suffix of length $k-1$ are connected by an edge. The contiguous reads are merged by finding the Eulerian path, which include every edge exactly once.

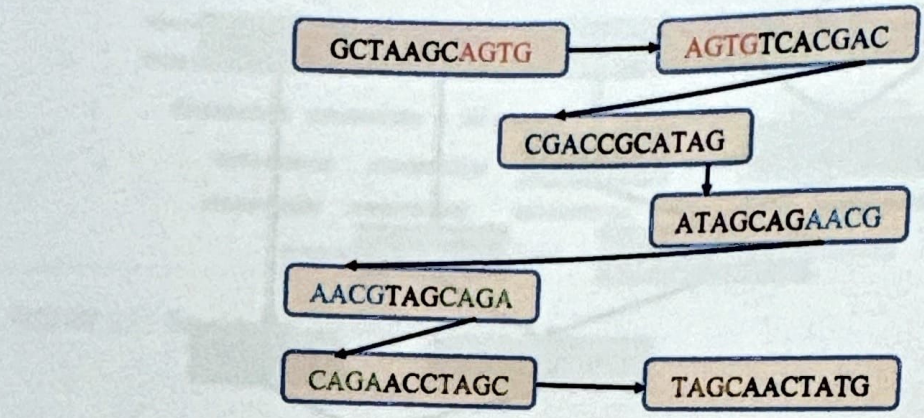


FIGURE 3.3 Overlap graphs and Hamiltonian path.

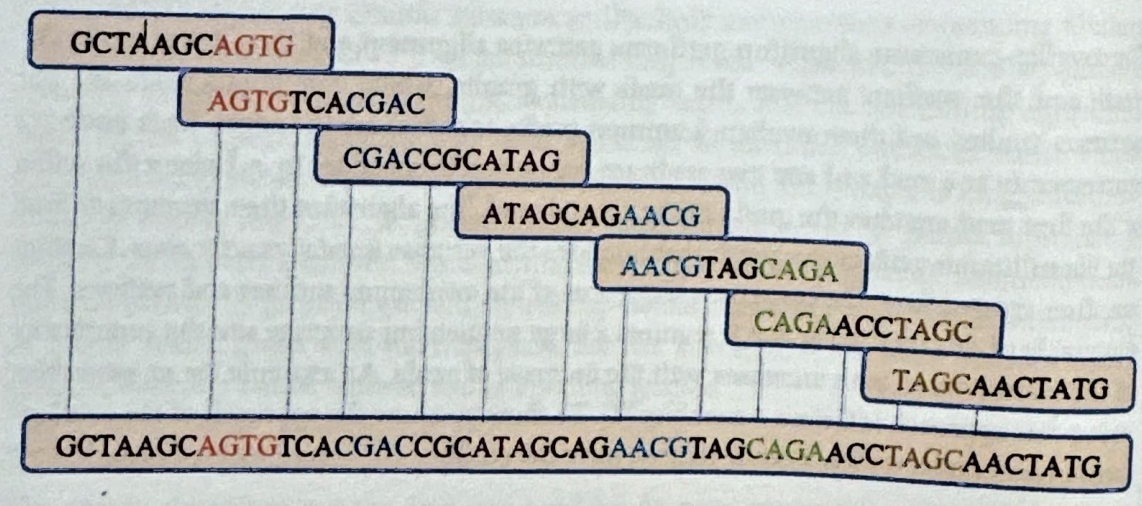


FIGURE 3.4 The overlaps and consensus sequence.

Figure 7: Overlap-consensus graphs

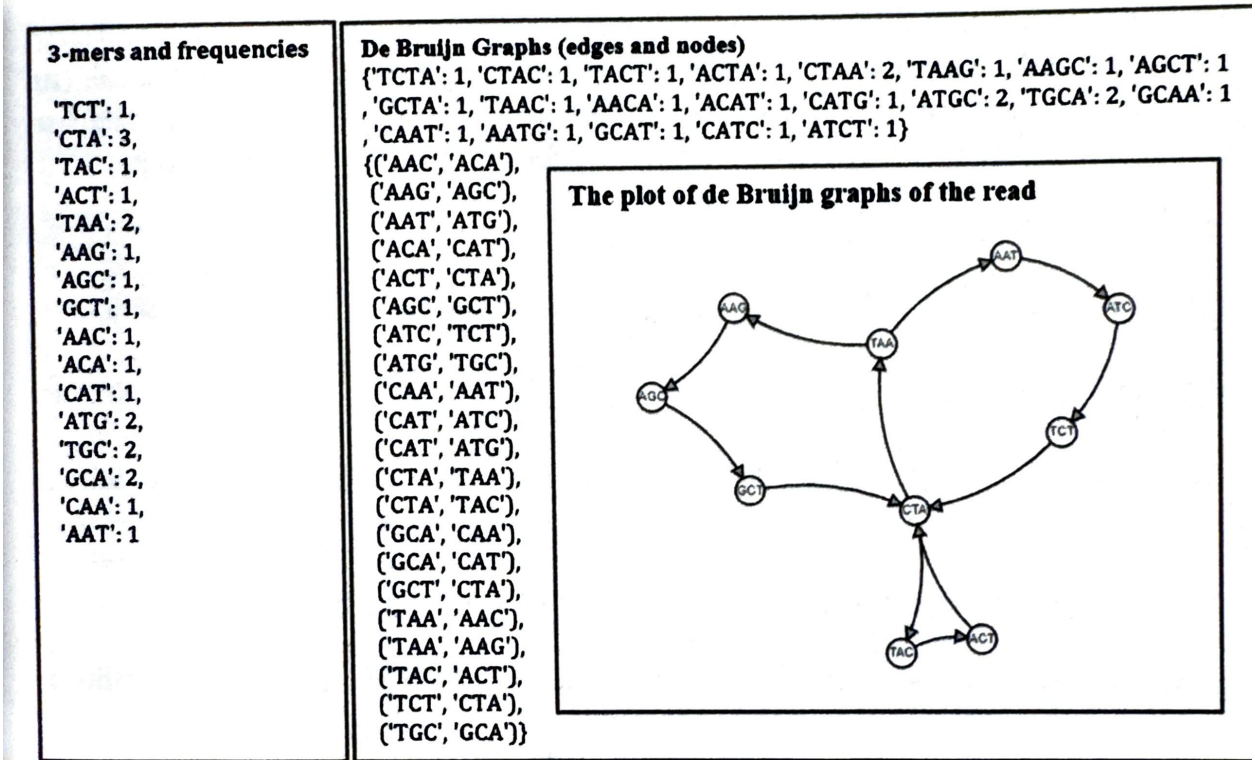
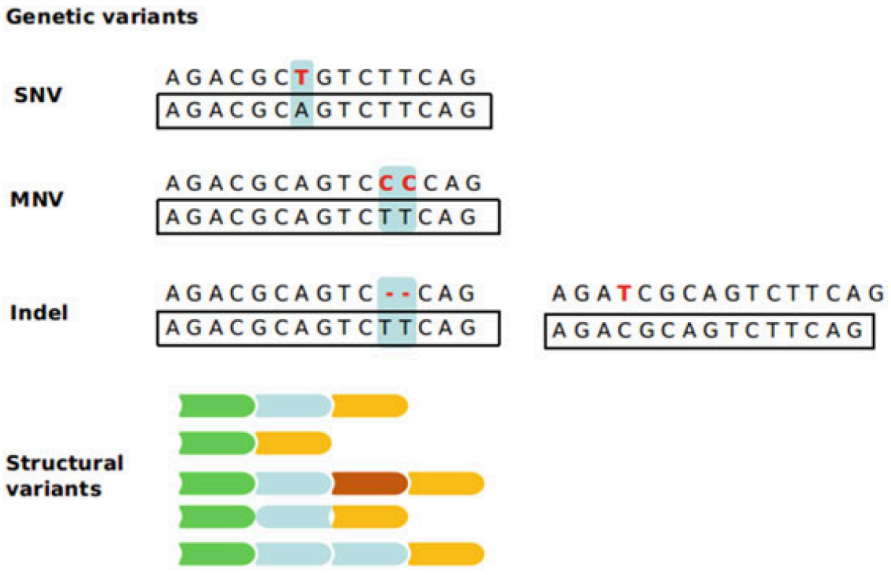


Figure 8: De Bruijn graphs

8 Genomic variants

8.1 Types of genomic variants



8.1.1 SNVs (single nucleotide variants):

also known as single base substitutions, are the simplest type of variation as they only involve the change of one base for another in a DNA sequence. These can be subcategorized into *transitions* (Ti) and *transversions* (Tv); *Ti*s are changes between two purines or between two pyrimidines, whereas the latter involve a change from a purine to a pyrimidine or vice versa. An example of a transition would be a G > A variant. If the SNV is common in a population (usually with an allele frequency > 1%), then it is referred to as a **SNP (single nucleotide polymorphism)**.

8.1.2 MNVs (multi-nucleotide variants):

which are sequence variants that involve the consecutive change of two or more bases. An example would be one of the types of mutations caused by UV irradiation, CC>TT. Similarly to SNVs, there are some MNVs that are found at higher frequencies in the population, which are referred to as **MNPs (multi-nucleotide polymorphism)**.

8.1.3 Indels (insertions and deletions):

which involve the gain or loss of one or more bases in a sequence. Usually, what is referred to as indel tends to be only a few bases in length. An example of a deletion would be CTGGT > C and an insertion would be represented as T > TGGAT.

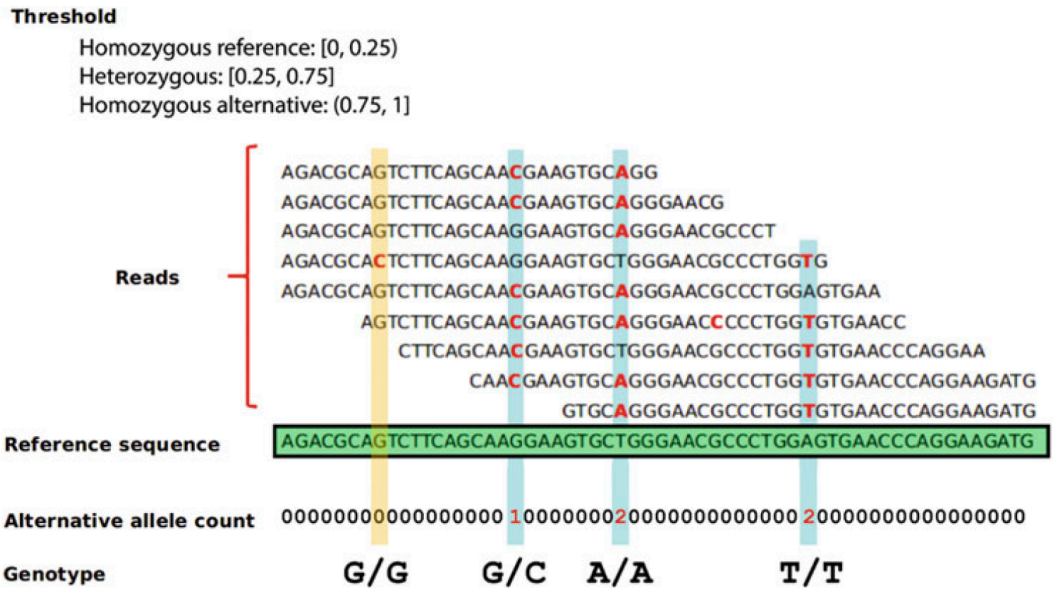
8.1.4 Structural variants:

which are genomic variations that involve larger segments of the genome. These can involve **inversions**, which is when a certain sequence in the genome gets reversed end to end, and **copy number variants** including amplifications, when a fraction of genome gets duplicated one or more times, and larger deletions, when large segments of the genome get lost. There is not a strict rule defining the number of base pairs that make the difference between an indel and a structural variant, but usually, a gain or loss of DNA would be called a structural variant if it involved more than one kilobase of sequence.

8.2 Methods of variant calling

8.2.1 Naive variant calling

A naive approach to determining the genotypes from a pile of sequencing reads mapped to a site in the genome is to count the number of reads with the reference and alternate alleles and to establish hard genotype thresholds.



In this method, reads are aligned to the reference sequence (green) and a threshold of the proportion of reads supporting each allele for calling genotypes is established (top). Then, at each position, the proportion of reads supporting the alternative allele is calculated and, based on the dosage of the alternative allele, a genotype is established. Yellow: a position where a variant is present but the proportion of alternative alleles does not reach the threshold ($1/6 < 0.25$).

8.2.2 Bayesian variant calling

The Bayesian approach includes information about the prior probability of a variant occurring at that site and the amount of information supporting each of the potential genotypes.

$$P(g|d) = \frac{P(d|g)P(g)}{P(d)}$$

where $P(d|g)$ is the likelihood of sequence data d given the genotype and $P(g)$ the prior probability of genotype g. The probabilities of a paticular genotype can be estimated over the whole sequence.

$P(d)$ can be computed as

$$P(d) = \sum_{g'} P(d|g')P(g')$$

8.2.3 Heuristic methods

these methods rely on heuristic quantities and algorithms to call a variant site, such as a minimum coverage and alignment quality thresholds, and stringent cut-offs like a minimum number of reads supporting a variant allele.

9 Applications

9.1 Extracting information in genomic regions of interest

Very often, one would want to know if a particular genomic feature falls in a particular genomic region of interest. This task can be handled very well by `GRanges` and `SummarizedExperiment` objects. We will look

at a few ways to creating these objects and a few ways we can manupulate them.

```
library(GenomicRanges)
library(rtracklayer)
library(SummarizedExperiment)
```

Create `GRanges` from different files: GFF, BED, and text files

```
get_granges_from_gff <- function(file_name) {
  gff <- rtracklayer::import.gff(file_name)
  as(gff, "GRanges")
}

get_granges_from_bed <- function(file_name){
  bed <- rtracklayer::import.bed(file_name)
  as(bed, "GRanges")
}

get_granges_from_text <- function(file_name){
  df <- readr::read_tsv(file_name, col_names = TRUE )
  GenomicRanges::makeGRangesFromDataFrame(df, keep.extra.columns = TRUE)
}

get_annotated_regions_from_gff <- function(file_name) {
  gff <- rtracklayer::import.gff(file_name)
  as(gff, "GRanges")
}
```

GFF file format: https://software.broadinstitute.org/software/igv/GFF

```
gr_from_gff <- get_annotated_regions_from_gff(file.path(getwd(), "arabidopsis_chr4.gff"))
gr_from_txt <- get_granges_from_text(file.path(getwd(), "arabidopsis_chr4.txt"))
head(gr_from_txt)
```

```
## GRanges object with 6 ranges and 2 metadata columns:
##       seqnames      ranges strand |        type  feature_id
##          <Rle>   <IRanges>  <Rle> | <character> <character>
##   [1]     Chr4   1180-1536      - |        gene   AT4G00005
##   [2]     Chr4  2895-10455      - |        gene   AT4G00020
##   [3]     Chr4 11355-13359      - |        gene   AT4G00026
##   [4]     Chr4 13527-14413      + |        gene   AT4G00030
##   [5]     Chr4 14627-16079      + |        gene   AT4G00040
##   [6]     Chr4 17792-20066      + |        gene   AT4G00050
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Extract genes and corresponding GRanges:

```
## Extract by seqname or metadata
genes_on_chr4 <- gr_from_gff[ gr_from_gff$type == "gene" & seqnames(gr_from_gff) %in% c("Chr4") ]
head(genes_on_chr4)
```

```
## GRanges object with 6 ranges and 10 metadata columns:
##       seqnames      ranges strand |   source     type     score     phase
##          <Rle>   <IRanges>  <Rle> | <factor> <factor> <numeric> <integer>
##   [1]     Chr4   1180-1536      - |   TAIR10     gene        NA      <NA>
##   [2]     Chr4  2895-10455      - |   TAIR10     gene        NA      <NA>
##   [3]     Chr4 11355-13359      - |   TAIR10     gene        NA      <NA>
##   [4]     Chr4 13527-14413      + |   TAIR10     gene        NA      <NA>
##   [5]     Chr4 14627-16079      + |   TAIR10     gene        NA      <NA>
##   [6]     Chr4 17792-20066      + |   TAIR10     gene        NA      <NA>
##                ID        Name                Note          Parent       Index
##       <character> <character>     <CharacterList> <CharacterList> <character>
##   [1]   AT4G00005   AT4G00005 protein_coding_gene                        <NA>
##   [2]   AT4G00020   AT4G00020 protein_coding_gene                        <NA>
##   [3]   AT4G00026   AT4G00026 protein_coding_gene                        <NA>
##   [4]   AT4G00030   AT4G00030 protein_coding_gene                        <NA>
##   [5]   AT4G00040   AT4G00040 protein_coding_gene                        <NA>
##   [6]   AT4G00050   AT4G00050 protein_coding_gene                        <NA>
##       Derives_from
##        <character>
##   [1]         <NA>
##   [2]         <NA>
##   [3]         <NA>
##   [4]         <NA>
##   [5]         <NA>
##   [6]         <NA>
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Manually creating region of interests and subset large region as `GRanges` object.

```
## By range, create synthetic ranges
region_of_interest_gr <- GRanges(
  seqnames = c("Chr4"),
  IRanges(c(10000), width= c(1000))
)

overlap_hits <- findOverlaps(region_of_interest_gr, genes_on_chr4)
features_in_region <- genes_on_chr4[subjectHits(overlap_hits) ]
features_in_region
```

```
## GRanges object with 1 range and 10 metadata columns:
##       seqnames     ranges strand |   source     type     score     phase
##          <Rle>  <IRanges>  <Rle> | <factor> <factor> <numeric> <integer>
##   [1]     Chr4 2895-10455      - |   TAIR10     gene        NA      <NA>
##                ID        Name                Note          Parent       Index
##       <character> <character>     <CharacterList> <CharacterList> <character>
##   [1]   AT4G00020   AT4G00020 protein_coding_gene                        <NA>
##       Derives_from
##        <character>
##   [1]         <NA>
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Create a random `SummarizedExperiment` that uses `GRanges` object and matrix of random data.

```
set.seed(4321)

experiment_counts <- matrix( runif(4308 * 6, 1, 100), 4308)
sample_names <- c(rep("ctrl",3), rep("test",3) )
se <- SummarizedExperiment::SummarizedExperiment(rowRanges = gr_from_txt, assays = list(experiment_coun
```

Let us find the overlapping regions in the experiments. Assay returns a subset of matrix data.

```
overlap_hits <- findOverlaps(region_of_interest_gr, se)
data_in_region <- se[subjectHits(overlap_hits) ]
assay(data_in_region)
```

```
##          [,1]     [,2]    [,3]     [,4]    [,5]     [,6]
## [1,] 91.00481 34.41582 42.7602 36.13053 47.6775 19.21672
```

9.2 Predicting open reading frames (ORFs) in long reference sequences

Often times, gene annotations are not usually available. Let us look a first stage pipline for finding potential genes and genomic loci of interest absolutely *de novo* and without information beyond the sequence.

Let us use a simple set of rules to find *open reading frames* - sequences that begin with a start codon and end with a stop codon. Will use `systemPipeR` package (to find ORF) and `GRanges` object that can be used for downstream analysis.

```
library(Biostrings)
library(systemPipeR) # to predict ORF
```

```
# Use arabidopsis_chloroplast genome sequence as input (from a fasta file)
dna_object <- readDNAStringSet(file.path(getwd(), "arabidopsis_chloroplast.fa"))

predicted_orfs <- predORF(dna_object, n='all', type='gr', mode='ORF', strand = 'both', longest_disjoint
predicted_orfs
```

```
## GRanges object with 2501 ranges and 2 metadata columns:
##           seqnames        ranges strand | subject_id inframe2end
##              <Rle>     <IRanges>  <Rle> |  <integer>   <numeric>
##      1 chloroplast   86762-93358      + |          1           2
##   1162 chloroplast     2056-2532      - |          1           3
##      2 chloroplast   72371-73897      + |          2           2
##   1163 chloroplast   77901-78362      - |          2           1
##      3 chloroplast   54937-56397      + |          3           3
##    ...         ...           ...    ... .        ...         ...
##   2497 chloroplast 129757-129762      - |       1336           3
##   2498 chloroplast 139258-139263      - |       1337           3
##   2499 chloroplast 140026-140031      - |       1338           3
##   2500 chloroplast 143947-143952      - |       1339           3
##   2501 chloroplast 153619-153624      - |       1340           3
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Compute the properties of reference genome

```
bases <- c("A", "C", "T", "G")
raw_seq_string <- strsplit(as.character(dna_object), "")

seq_length <- width(dna_object[1])
counts <- letterFrequency(dna_object[1], letters = c("A", "T", "G", "C"))
probs <- unlist(lapply(counts, function(base_count){signif(base_count / seq_length, 2) }))
dna_object[1]
```

```
## DNAStringSet object of length 1:
##      width seq                                                 names
## [1] 154478 ATGGGCGAACGACGGGAATTGAA...TAATAACTTGGTCCCGGGCATC chloroplast
```

```
probs
```

```
## [1] 0.31 0.32 0.18 0.18
```

Find the longest ORF length of the synthetic genome and keep the ORF which is longer than that in the given genome:

```
# write a function to return longest ORF
get_longest_orf_in_random_genome <- function(x,
                                             length = 1000,
                                             probs = c(0.25, 0.25, 0.25, 0.25),
                                             bases = c("A","C","T","G"))
{
  random_genome <- paste0(sample(bases, size = length, replace = TRUE, prob = probs), collapse = "")
  random_dna_object <- DNAStringSet(random_genome)
  names(random_dna_object) <- c("random_dna_string")
  orfs <- predORF(random_dna_object, n = 1, type = 'gr', mode='ORF', strand = 'both', longest_disjoint =
  return(max(width(orfs)))
}

# generate 10 simulated random genomes
random_lengths <- unlist(lapply(1:10, get_longest_orf_in_random_genome, length = seq_length, probs = pr

# get length of longest random ORF
longest_random_orf <- max(random_lengths)

# Keep only the predicted ORF longer than the longest random ORF
keep <- width(predicted_orfs) > longest_random_orf
orfs_to_keep <- predicted_orfs[keep]
orfs_to_keep
```

```
## GRanges object with 10 ranges and 2 metadata columns:
##         seqnames        ranges strand | subject_id inframe2end
##            <Rle>     <IRanges>  <Rle> |  <integer>   <numeric>
##    1 chloroplast   86762-93358      + |          1           2
##    2 chloroplast   72371-73897      + |          2           2
##    3 chloroplast   54937-56397      + |          3           3
##    4 chloroplast   57147-58541      + |          4           1
##    5 chloroplast   33918-35141      + |          5           1
##    6 chloroplast   32693-33772      + |          6           2
```

```
##    7 chloroplast 109408-110436      + |          7           3
##    8 chloroplast 114461-115447      + |          8           2
##    9 chloroplast 141539-142276      + |          9           2
##   10 chloroplast   60741-61430      + |         10           1
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

```
##writing the resutls to a file in fasta format
extracted_orfs <- BSgenome::getSeq(dna_object, orfs_to_keep)
names(extracted_orfs) <- paste0("orf_", 1:length(orfs_to_keep))
writeXStringSet(extracted_orfs, "saved_orfs.fa")
```

9.3 Finding SNPs and indels from sequence data using `VariantTools`

```
library(GenomicRanges) # handles genomic locations within a genome
library(gmapR) # align short-range data
library(rtracklayer) #interface to genome annotation files and the UCSC genome browser
library(VariantAnnotation)
library(VariantTools)
```

A key task in sequence analysis is to take an alignment of high-throughput sequences stored in BAM file (.bam) and compute a list of variant positions. Note that after FASTAQ file alignment to the reference genome, the outputs appear in BAM files. The results of variant calling is usually stored in a VCF file of variants.

We will use a set of synthetic reads from human genome chromosome 17. Let us first read `.bam` and `.fa` files.

```
bam_folder <- file.path(getwd())

# reference genome
bam_file <- file.path( bam_folder, "hg17_snps.bam")

# testing genome
fasta_file <- file.path(bam_folder,"chr17.83k.fa")
```

Create a `GmapGenome` object

```
fa <- rtracklayer::FastaFile(fasta_file)
genome <- gmapR::GmapGenome(fa, create=TRUE)
```

Create a parameter object and call the Varints

```
qual_params <- TallyVariantsParam(genome = genome, minimum_mapq = 20)
var_params <- VariantCallingFilters(read.count = 19, p.lower = 0.01)

called_variants <- callVariants(bam_file, qual_params, calling.filters = var_params)
head(called_variants)
```

```
## VRanges object with 6 ranges and 17 metadata columns:
```

```
##           seqnames    ranges strand         ref              alt     totalDepth
##              <Rle> <IRanges>  <Rle> <character> <characterOrRle> <integerOrRle>
##   [1] NC_000017.10        64      *           G               T            759
##   [2] NC_000017.10        69      *           G               T            812
##   [3] NC_000017.10        70      *           G               T            818
##   [4] NC_000017.10        73      *           T               A            814
##   [5] NC_000017.10        77      *           T               A            802
##   [6] NC_000017.10        78      *           G               T            798
##            refDepth       altDepth   sampleNames softFilterMatrix | n.read.pos
##       <integerOrRle> <integerOrRle> <factorOrRle>         <matrix> |  <integer>
##   [1]            739             20          <NA>                  |         17
##   [2]            790             22          <NA>                  |         19
##   [3]            796             22          <NA>                  |         20
##   [4]            795             19          <NA>                  |         13
##   [5]            780             22          <NA>                  |         19
##   [6]            777             21          <NA>                  |         17
##       n.read.pos.ref raw.count.total count.plus count.plus.ref count.minus
##            <integer>       <integer>  <integer>      <integer>   <integer>
##   [1]             64             759         20            739           0
##   [2]             69             812         22            790           0
##   [3]             70             818         22            796           0
##   [4]             70             814         19            795           0
##   [5]             70             802         22            780           0
##   [6]             70             798         21            777           0
##       count.minus.ref count.del.plus count.del.minus read.pos.mean
##             <integer>      <integer>       <integer>     <numeric>
##   [1]               0              0               0       30.9000
##   [2]               0              0               0       40.7273
##   [3]               0              0               0       34.7727
##   [4]               0              0               0       36.1579
##   [5]               0              0               0       38.3636
##   [6]               0              0               0       39.7143
##       read.pos.mean.ref read.pos.var read.pos.var.ref     mdfne mdfne.ref
##                <numeric>    <numeric>        <numeric> <numeric> <numeric>
##   [1]           32.8755      318.558          347.804        NA        NA
##   [2]           35.4190      377.004          398.876        NA        NA
##   [3]           36.3442      497.762          402.360        NA        NA
##   [4]           36.2176      519.551          402.843        NA        NA
##   [5]           36.0064      472.327          397.070        NA        NA
##   [6]           35.9241      609.076          390.463        NA        NA
##       count.high.nm count.high.nm.ref
##           <integer>         <integer>
##   [1]            20               738
##   [2]            22               789
##   [3]            22               796
##   [4]            19               769
##   [5]            22               780
##   [6]            21               777
##   -------
##   seqinfo: 1 sequence from chr17.83k genome
##   hardFilters(4): nonRef nonNRef readCount likelihoodRatio
```

Write VCF file

```
VariantAnnotation::sampleNames(called_variants) <- "sample_name"
vcf <- VariantAnnotation::asVCF(called_variants)
VariantAnnotation::writeVcf(vcf, "hg17.vcf")
```

Load the annotations and feature positions

```
get_annotated_regions_from_gff <- function(file_name) {
  gff <- rtracklayer::import.gff(file_name)
  as(gff, "GRanges")
}

get_annotated_regions_from_bed <- function(file_name){
  bed <- rtracklayer::import.bed(file_name)
  as(bed, "GRanges")
}

genes <- get_annotated_regions_from_gff(file.path( bam_folder, "chr17.83k.gff3"))
```

calculate which variants overlap with genes and subest the genes

```
overlaps <- GenomicRanges::findOverlaps(called_variants, genes)

genes[subjectHits(overlaps)][1:6]
```

```
## GRanges object with 6 ranges and 20 metadata columns:
##           seqnames      ranges strand |   source       type     score     phase
##              <Rle>   <IRanges>  <Rle> | <factor>   <factor> <numeric> <integer>
##   [1] NC_000017.10 64099-76866      - |   havana ncRNA_gene        NA      <NA>
##   [2] NC_000017.10 64099-76866      - |   havana lnc_RNA           NA      <NA>
##   [3] NC_000017.10 64099-65736      - |   havana exon              NA      <NA>
##   [4] NC_000017.10 64099-76866      - |   havana ncRNA_gene        NA      <NA>
##   [5] NC_000017.10 64099-76866      - |   havana lnc_RNA           NA      <NA>
##   [6] NC_000017.10 64099-65736      - |   havana exon              NA      <NA>
##                           ID            Name     biotype      description
##                  <character>     <character> <character>      <character>
##   [1]   gene:ENSG00000280279      AC240565.2     lincRNA novel transcript
##   [2] transcript:ENST00000..  AC240565.2-201     lincRNA             <NA>
##   [3]                   <NA> ENSE00003759547        <NA>             <NA>
##   [4]   gene:ENSG00000280279      AC240565.2     lincRNA novel transcript
##   [5] transcript:ENST00000..  AC240565.2-201     lincRNA             <NA>
##   [6]                   <NA> ENSE00003759547        <NA>             <NA>
##               gene_id  logic_name     version                Parent
##           <character> <character> <character>        <CharacterList>
##   [1] ENSG00000280279      havana           1
##   [2]            <NA>        <NA>           1   gene:ENSG00000280279
##   [3]            <NA>        <NA>           1 transcript:ENST00000..
##   [4] ENSG00000280279      havana           1
##   [5]            <NA>        <NA>           1   gene:ENSG00000280279
##   [6]            <NA>        <NA>           1 transcript:ENST00000..
##               tag   transcript_id transcript_support_level constitutive
##       <character>     <character>              <character>  <character>
##   [1]        <NA>            <NA>                     <NA>         <NA>
```

```
##   [2]       basic ENST00000623180                      5         <NA>
##   [3]        <NA>            <NA>                     <NA>            1
##   [4]        <NA>            <NA>                     <NA>         <NA>
##   [5]       basic ENST00000623180                      5         <NA>
##   [6]        <NA>            <NA>                     <NA>            1
##       ensembl_end_phase ensembl_phase         exon_id        rank
##             <character>   <character>     <character> <character>
##   [1]              <NA>          <NA>            <NA>        <NA>
##   [2]              <NA>          <NA>            <NA>        <NA>
##   [3]                -1            -1 ENSE00003759547           5
##   [4]              <NA>          <NA>            <NA>        <NA>
##   [5]              <NA>          <NA>            <NA>        <NA>
##   [6]                -1            -1 ENSE00003759547           5
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

9.4 Ploting features of genetic maps

Often, we want to see on a chromosome or genetic map where some features of interest lie in relation to others. These plots are called *chromosome plots* or *ideograms*, and we will see how `karyoploteR` can do.

```
library(karyoploteR)
library(GenomicRanges)
```

Create a `GRanage` object:

```
genome_df <- data.frame(
  chr = paste0("chr", 1:5),
  start = rep(1, 5),
  end = c(34964571, 22037565, 25499034, 20862711, 31270811)
)
genome_gr <- makeGRangesFromDataFrame(genome_df)
genome_gr
```

```
## GRanges object with 5 ranges and 0 metadata columns:
##       seqnames     ranges strand
##          <Rle>  <IRanges>  <Rle>
##   [1]     chr1 1-34964571      *
##   [2]     chr2 1-22037565      *
##   [3]     chr3 1-25499034      *
##   [4]     chr4 1-20862711      *
##   [5]     chr5 1-31270811      *
##   -------
##   seqinfo: 5 sequences from an unspecified genome; no seqlengths
```

Set up SNPs positions we draw as markers

```
snp_pos <- sample(1:1e7, 25)
snps <- data.frame(
  chr = paste0("chr", sample(1:5,25, replace=TRUE)),
  start = snp_pos,
  end = snp_pos
```

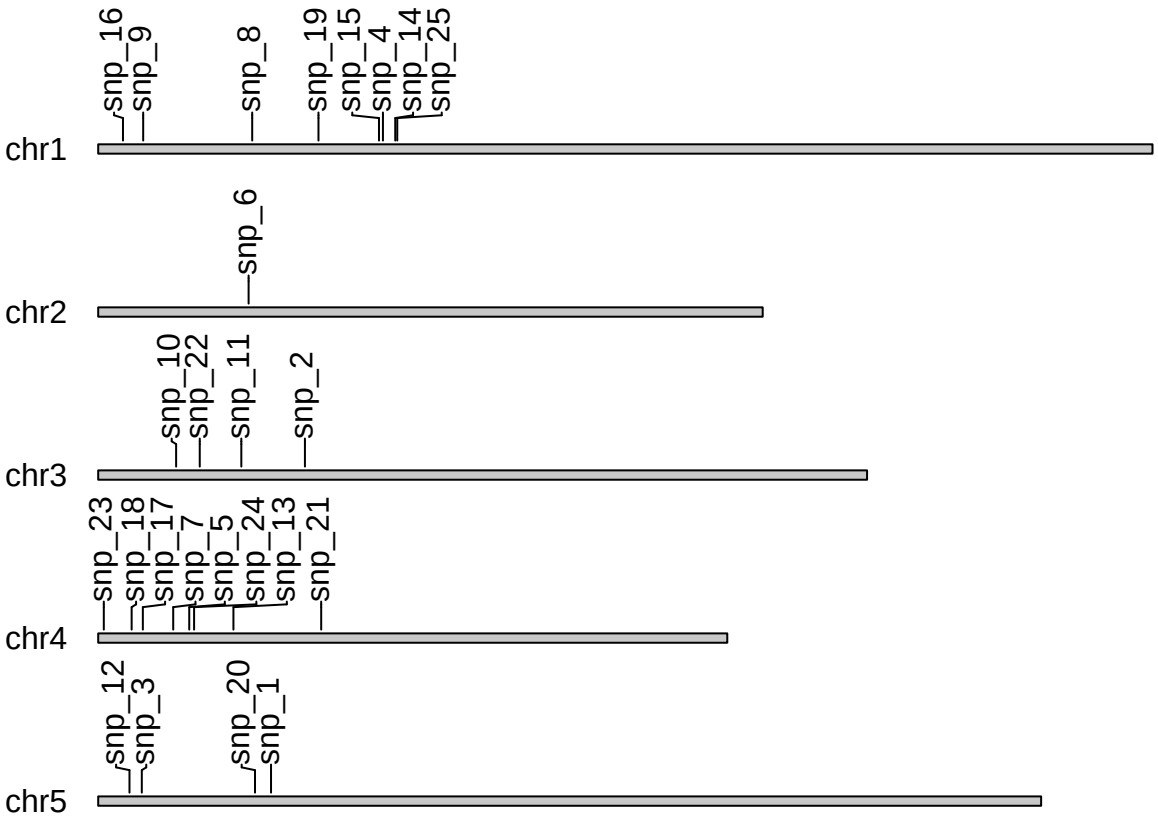
```
)
snps_gr <- makeGRangesFromDataFrame(snps)
snp_labels <- paste0("snp_", 1:25)

snps_gr
```

```
## GRanges object with 25 ranges and 0 metadata columns:
##        seqnames    ranges strand
##           <Rle> <IRanges>  <Rle>
##    [1]     chr5   5731177      *
##    [2]     chr3   6855485      *
##    [3]     chr5   1445614      *
##    [4]     chr1   9442378      *
##    [5]     chr4   3018935      *
##    ...      ...       ...    ...
##   [21]     chr4   7395203      *
##   [22]     chr3   3367345      *
##   [23]     chr4    185686      *
##   [24]     chr4   3180060      *
##   [25]     chr1   9916951      *
##   -------
##   seqinfo: 5 sequences from an unspecified genome; no seqlengths
```

```
plot.params <- getDefaultPlotParams(plot.type=1)
plot.params$data1outmargin <- 600

kp <- plotKaryotype(genome=genome_gr, plot.type = 1, plot.params = plot.params)
kpPlotMarkers(kp, snps_gr, labels = snp_labels)
```



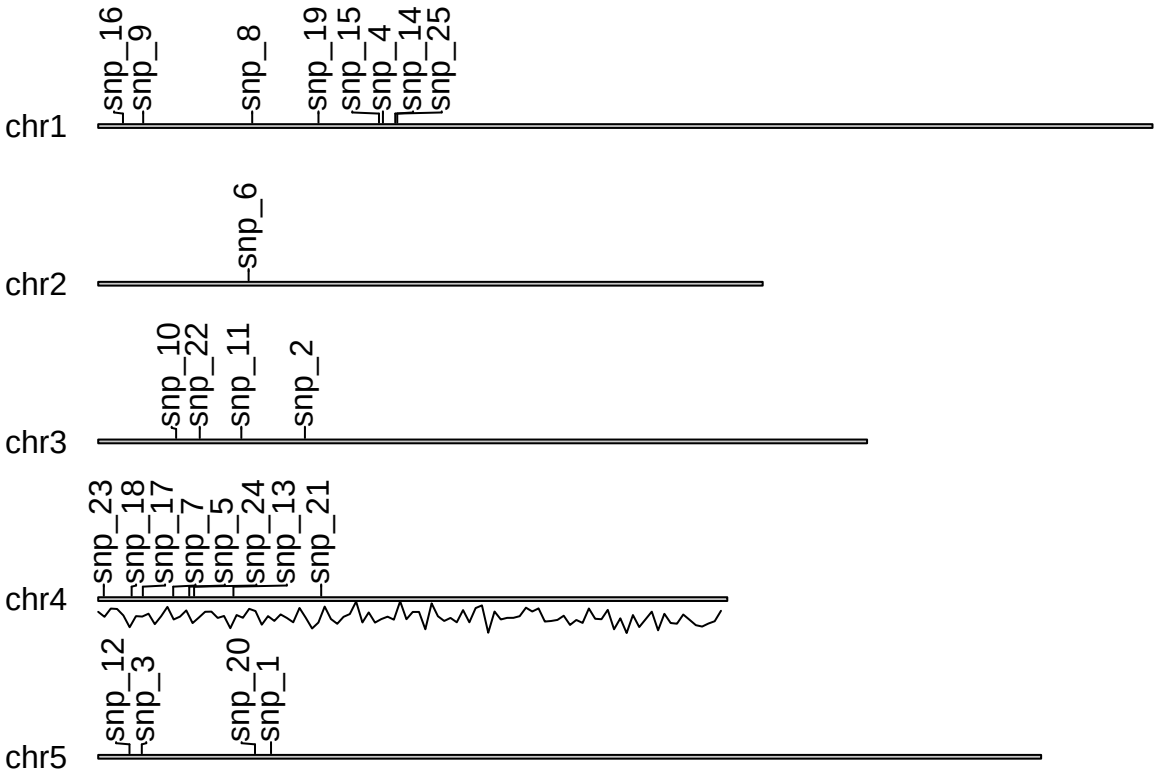
We can add some numeric data to the plot. For example, lets introduce some random plot for chr4:

```
### create random data to chr4
numeric_data <- data.frame(
  y = rnorm(100,mean = 1,sd = 0.5  ),
  chr = rep("chr4", 100),
  start = seq(1,20862711, 20862711/100),
  end = seq(1,20862711, 20862711/100)
)

numeric_data_gr <- makeGRangesFromDataFrame(numeric_data)

plot.params <- getDefaultPlotParams(plot.type=2)
plot.params$data1outmargin <- 800
plot.params$data2outmargin <- 800
plot.params$topmargin <- 800

kp <- plotKaryotype(genome=genome_gr, plot.type = 2, plot.params = plot.params)
kpPlotMarkers(kp, snps_gr, labels = snp_labels)
kpLines(kp, numeric_data_gr, y = numeric_data$y, data.panel=2)
```



9.5 Estimating the copy number at a locus of interest

We often want to know if a locus has been duplicated or its **copy number** has increased. Our approach is to use DNA-seq read coverage after alignment to estimate a background level of coverage and then inspect the coverage on our region of interest. The ratio of coverage give an estimate of the copy number of the region.

```
library(csaw) # Bioconductor package for find CNV
```

Get counts in windows across the hg17 genome:

```
whole_genome <- csaw::windowCounts(
  file.path(getwd(), "hg17_snps.bam"), # the bam file contains counts
  bin = TRUE,
  filter = 0,
  width = 100,
  param = csaw::readParam(
    minq = 20,
    dedup = TRUE,
    pe = "both"
  )
)
colnames(whole_genome) <- c("h17")
```

Extract the data from `SummarizedExperiment` object. Set a threshold and make lower counts to NA.

```
counts <- assay(whole_genome)[,1]

min_count <- quantile(counts, 0.1)[[1]]
counts[counts < min_count] <- NA # lowest counts to NA
```

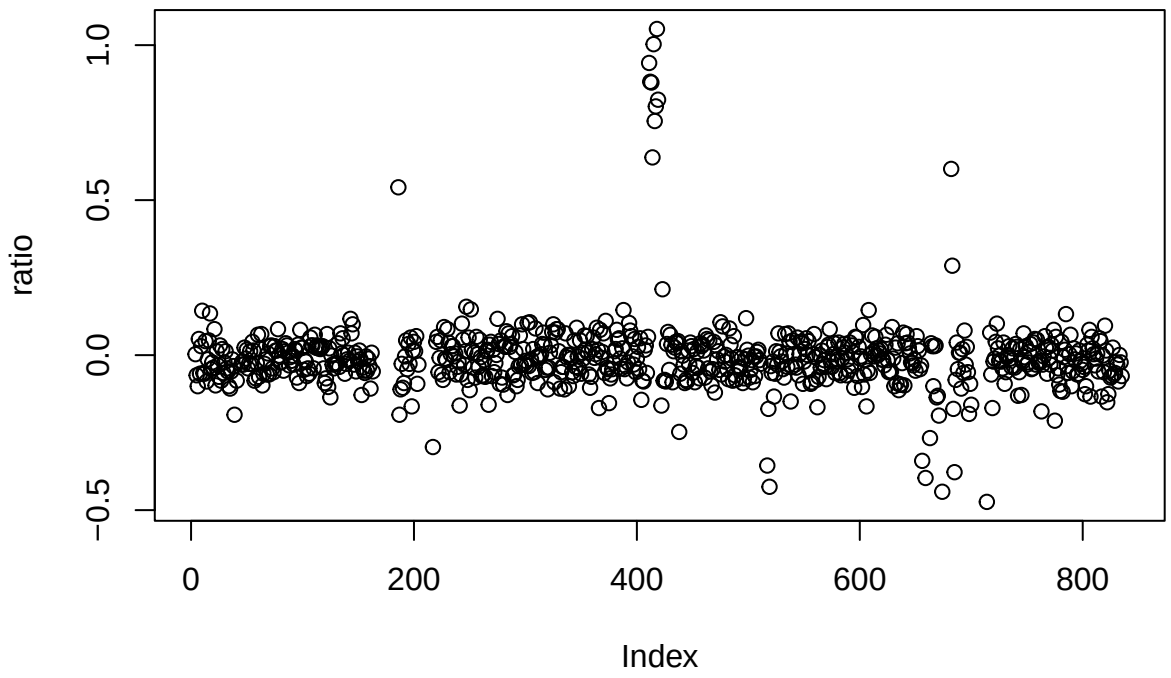
Calculate the mean coverage and ratio in each window to that mean coverage, and inspect the ratio vector with a plot.

```
n <- length(counts)
doubled_windows <- 10

left_pad <- floor( (n/2) - doubled_windows )
right_pad <- n - left_pad -doubled_windows
multiplier <- c(rep(1, left_pad ), rep(2,doubled_windows), rep(1, right_pad) )
counts <- counts * multiplier

mean_cov <- mean(counts, na.rm=TRUE)

ratio <- matrix(log2(counts / mean_cov), ncol = 1)
plot(ratio)
```



Build a `SummarisedExperiment` with new data

```
se <- SummarizedExperiment(assays=list(ratio), rowRanges= rowRanges(whole_genome), colData = c("Coverag
```

Create a region of interest and extract the coverage from it

```
region_of_interest <- GRanges(
  seqnames = c("NC_000017.10"),
  IRanges(c(40700), width = c(1500) )
)
```

```
overlap_hits <- findOverlaps(region_of_interest, se)
data_in_region <- se[subjectHits(overlap_hits)]
assay(data_in_region)
```

```
##              [,1]
##  [1,]  0.01725283
##  [2,]  0.03128239
##  [3,] -0.05748994
##  [4,]  0.05893873
##  [5,]  0.94251006
##  [6,]  0.88186246
##  [7,]  0.87927929
##  [8,]  0.63780103
##  [9,]  1.00308550
## [10,]  0.75515798
## [11,]  0.80228189
## [12,]  1.05207419
## [13,]  0.82393626
## [14,]          NA
## [15,]          NA
## [16,] -0.16269298
```

9.6 Finding phenotype and genotype associations with GWAS

Genome-wide association studies (GWAS) of genotype and phenotypes is a powerful application to find the presence of gentic variants in many samples of high-throughput sequencing data. GWAS is a genomic analysis of gentic variants in different individuals of genetic lines to see any particular variant is associated with a trait in a large population.

There are various approaches for doing GWAS, which rely on gathering data on vaiants in particular samples and working out each sample's genotype before cross-referencing with the phenotype in some way or other. We will use the `GWAS` function from `rrBLUP` package.

```
library(VariantAnnotation)
library(rrBLUP)
set.seed(1234) # for reproducibility
```

Get the VCF (Variant Call Format) file

```
vcf_file <- file.path(getwd(), "small_sample.vcf")
```

Extract the genotype, sample, and marker position information

```
vcf <- readVcf(vcf_file, "hg19")
header(vcf)
```

```
## class: VCFHeader
## samples(3): NA00001 NA00002 NA00003
## meta(3): fileformat reference contig
## fixed(1): FILTER
## info(3): DP AF DB
## geno(2): GT DP
```

```
samples <- samples(header(vcf)) # take a sample
samples
```

```
## [1] "NA00001" "NA00002" "NA00003"
```

```
chrom <- as.character(seqnames(rowRanges(vcf)))
chrom
```

```
## [1] "20" "20" "20"
```

```
pos <- as.numeric(start(rowRanges(vcf)))
pos
```

```
## [1]   14370   17330 1230237
```

```
gts <- geno(vcf)$GT
gts
```

```
##                NA00001 NA00002 NA00003
## rs6054257      "0/0"   "0/1"   "1/1"
## 20:17330_T/A   "0/0"   "0/1"   "0/0"
## 20:1230237_T/G "0/0"   "0/0"   "1/0"
```

```
markers <- rownames(gts)
```

Convert VCF genotypes into the convention used by the GWAS function:

```
convert <- function(v){
  v <- gsub("0/0", 1, v)
  v <- gsub("0/1", 0, v)
  v <- gsub("1/0", 0, v)
  v <- gsub("1/1",-1, v)
  return(v)
}
```

Call the function and convert the result into a numeric matrix to map heterozygous and homozygous annotations to that used in GWAS.

```
gt_char<- apply(gts, convert, MARGIN = 2)

genotype_matrix <- matrix(as.numeric(gt_char), nrow(gt_char) )
colnames(genotype_matrix)<- samples
genotype_matrix
```

```
##      NA00001 NA00002 NA00003
## [1,]       1       0      -1
## [2,]       1       0       1
## [3,]       1       1       0
```

Build a dataframe describing the variant

```
variant_info <- data.frame(marker = markers, chrom = chrom, pos = pos)
```

Build a variant/genotype dataframe

```
genotypes <-  cbind(variant_info, as.data.frame(genotype_matrix))
genotypes
```

```
##           marker chrom     pos NA00001 NA00002 NA00003
## 1      rs6054257    20   14370       1       0      -1
## 2   20:17330_T/A    20   17330       1       0       1
## 3 20:1230237_T/G    20 1230237       1       1       0
```

Build a phenotype dataframe

```
phenotypes <- data.frame(
  line = samples,
  score = rnorm(length(samples))
)
phenotypes
```

```
##      line      score
## 1 NA00001 -1.2070657
## 2 NA00002  0.2774292
## 3 NA00003  1.0844412
```

Run GWAS to map genotypes and phenotypes:

```
GWAS(phenotypes, genotypes, plot=FALSE)
```

```
## [1] "GWAS for trait: score"
## [1] "Variance components estimated. Testing markers."
```

```
##           marker chrom     pos     score
## 1      rs6054257    20   14370 0.3010543
## 2   20:17330_T/A    20   17330 0.3010057
## 3 20:1230237_T/G    20 1230237 0.1655498
```

Returns a data frame where the first three columns are the marker name, chromosome, and position, and subsequent columns are the marker scores (-log_{10}p) for the traits.

10 References:

- R Bioinformatics Cookbook by Dan MacLean:
  https://github.com/PacktPublishing/R-Bioinformatics-Cookbook
- Next Generation Sequencing And Data Analysis edited by Melanie Kappelmann-Fenzl:
  https://link.springer.com/book/10.1007/978-3-030-62490-3