SCHOOL OF ELECTRICAL AND INFORMATION ENGINEERING

UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

School of Electrical and Information Engineering

COMS4040 – High performance computing and scientific data management

Assignment 1 – version 1.1

15 March 2022

Introduction

The goal of this assignment is to (a) give you experience in Rust; (b) make you consider architectural and algorithm concepts in writing code which must run fast.

Background

You will write code that manipulates *genotype* data. Genotype data represents a sub-set of genetic variants in an individual or a set of individuals. For our purposes we will consider a copy of the genome to be a vector over the alphabet $\{A, C, G, T, X\}$ where A, C, G, and T represent molecules that make up the DNA and X represents missing information (in the sense that there is data at that position, but for some reason we don't know that it is). We will also make the assumption that for the data *we* will be storing that in the human population (or whatever animal or plant we are studying) that at each position there exist exactly two possibilities of what may be found in the population we are studying.

At each position in the genome (or at least for the vast bulk of the genome), an individual has two copies of genome – one inherited from the mother, one from the father.

Some jargon we will use are:

- *allele* – these are the variants that are found at each position. Within a specific species in most positions only one variant will be found. In some cases, there will be two alleles (and in some cases – which we will not consider – more alleles can be found).

  E.g., in humans in the BRCA2 gene on chromosome 13 position 32,328,640 there are two alleles A or G. Just over 99% of humans have two copies of the G allele. About 0.6% of individuals have one copy of an A and one copy of a G, and a tiny number have two copies of the G. But at position 32,328,641 there is only one allele – all humans who have ever been genotyped have two copies of T. In all *our* data at each position there are exactly two alleles possible.
- In standard databases, one of the alleles is designated as the *reference*. This is usually, but not always, the most commonly found one. In the above example at position 32,328,640 the G allele is the reference allele. The other allele is called the *alternate* allele (some tools including PLINK refer to these as the major and minor allele – it means the same thing).
- The alternate allele frequency at a position is the proportion of times the alternate allele is seen. As each individual has two copies of the genome each position may have two copies of the reference allele (we call this *homozygous* in the reference allele), one copy of each allele (this is called *heterozygous*) or two copies of the alternate allele (*homozygous* in the alternate allele).
- We call each position for which we record a genotype a *SNP* (pronounced "snip") important here).

PLINK format

We will be using a format called PLINK, described in more detail below. PLINK uses a compact encoding, and for each individual and each position, PLINK encodes a specific genotype with

- 00: if an individual is homozygous in the alternate allele;
- 01: if an individual is heterozygous;
- 10: if an the genotype is unknown / data about the individual at that position is missing; and
- 11: of an individual is homozygous in the reference allele.

PLINK is a widely used bioinformatics program, and its file format is used by many other programs as way of compactly representing genotype data. A PLINK data set comprises three files, typically with the same base name and different suffixes. Suppose our data set is called *example*.

- The *fam* file (in this case `example.fam`) describes the individuals – it has the identities of the individuals, family relationship, biological sex, and in some cases disease status. For our purposes, the only thing that will be important is the number of individuals that there are in the data set (the number of lines in the file).
- The *bim* file (in this case `example.bim`) describes the SNPs: where on the genome the SNP is found, the ID of a SNP in a standard database, and what the reference and alternate alleles are. For our purposes, the only thing that will be important is the number of SNPs (the number of lines in the file).
- The *bed* file (in this case `example.bed`) which stores the actual genotype data, essentially as a two dimensional array, as explained below. Note that the size of the *bed* file is proportional to the product of the number of SNPs and the number of individuals and so is typically orders of magnitude bigger than the other files.
- A typical case would be – number of SNPs $\approx$ 2 million, and the number of individuals $\approx$ 10k.

In a specific data set that the SNP data that we have about the individuals is common – that is we know the data (except for a small set of missing data) about the same positions for all individuals. The structure of the *bed* file is as follows

- There are three bytes called the *magic number*[1]. The first two bytes are 0x6C and 0x1B.

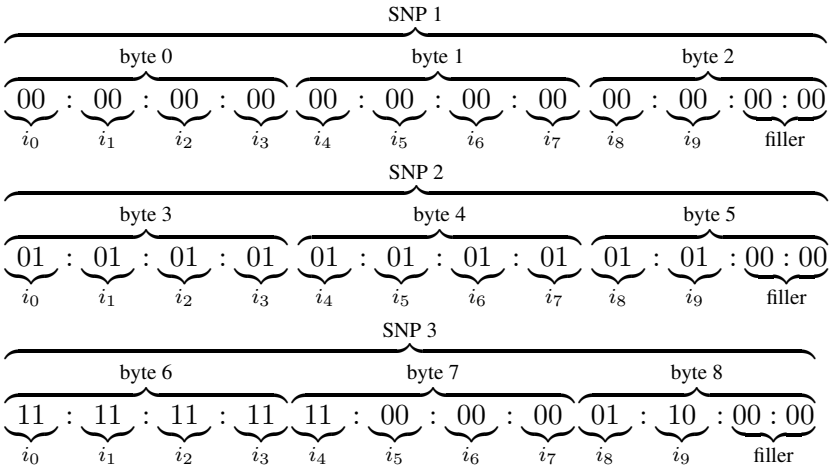
  The third byte is 0 or 1 and tells whether the data is organised by SNP (1) or by individual (0). SNP-major (1) is the default and described below.

- The rest of the file has the actual genotype data encoded as described above, storing four genotypes in a byte. In SNP-major format, we first store all the genotypes of the all the individuals for first SNP (four individuals per byte), followed by all the genotypes of the all the individuals for the next SNP and so on. If the number of individuals is not a multiple of 4 then in the last byte we only use the number of bits that are necessary and ignore the remaining bits. This means that for each SNP, the genotypes start at the beginning of a byte.

  If there are N individuals we need $\lceil N/4 \rceil$ bytes for each SNP, and so if there are M SNPs, we need $M \times \lceil N/4 \rceil$ bytes in total. You should be able to see why the *bim* and *fam* files are needed even if we ignore the contents – without knowing M and N we have no way of interpreting the *bed* file. If there are 144 data bytes in the file then it could be organised as 144×1, 72×2, 48×3, 36×4 and so on...

[1]Note that the use of magic numbers is common as it allows programs that manipulate data to validate the files they are processing are of the right type or to handle slightly different versions of the same format. See `https://en.wikipedia.org/wiki/List_of_file_signatures` for more examples

Example: Suppose we have 10 individuals in the study ($i_0, \ldots i_1 0$) and three SNPs. For the first SNP all individuals are homozygous in the alternate allele; for the second SNP all individuals are heterozygous; for the third SNP, the first five individuals are homozygous in the reference allele, the next three individuals are homozygous in the alternare allele, followed by someone who is heterozygous with the last person having missing data. This requires 9 bytes in all. Each SNP requires 3 bytes, with the last 4 bits of the third byte being unused. This is represented below. For visual convenience I separate the two bits that make up each genotype in a byte with a colon. I include a line-break in the visual representation to fit on the physical page, but remember the computer just stores all the bytes in one long sequence.



You can find more detail of the PLINK format at `https://www.cog-genomics.org/plink/1.9/formats`.

Endianness

You need to be aware of the endianness of the representation of PLINK bytes. In the positional numeral system we use, on paper we represent digits (base 10) or bits (base 2) with the most significant bit or digit on the left, with less significant digits/bits as we move rightwards. Left/right are human visual representations, not computer memory terms. If we just think of a byte (the same issue also generalises to several bytes making a word, but that's not an issue for us here), there are two sensible conventions to make: (i) we could have the most significant bit of our number stored in bit 0 and the least significant in bit 7; or (ii) we could have the least significant bit in bit 0 and the most significant bit in bit 7. The former convention is called "big-endian" and the latter is "little-endian". There are arguments for both – not our concern here [2] – we just need to know which convention PLINK uses. This is independent of the endian-ness convention that your computer's architecture uses – e.g, x86_64 or AMD. I leave determining PLINK's endianness as an exercise to determine so you fully understand. This is important to get right because depending on PLINK's convention, what we *conceptually* think of as 10 (missing) could have the 1 in a higher or lower numbered bit than the 0. Without knowing the endian-ness of *PLINK* you can't know if the pattern 01010101 in a byte is a set of four heterozygous or four missing genotypes, and you won't know whether the pattern 11110000 is two homozygous reference genotypes followed by two homozygous alternate genotypes or vice-versa[3].

[2] See `https://en.wikipedia.org/wiki/Endianness` for more detail if interested.

[3] the other way around

Notes

- The bim and fam files are human readable text files and can be read in using functions that support text, line-oriented files. The bed files are not human readable – your code must read in and interpret streams of bytes.
- The alternate allele frequency is measured as a proportion of non-missing alleles which are alternate.

$$aaf_i = \frac{\sum_{j=0}^{N-1} aac_{ij}}{2(N - m_i)},$$

where aaf_i is alternate allele frequency of SNP i, m_i is the number of missing genotypes there are for SNP i and aac_{ij} is the number/count of alternate alleles that individual j has at position i (2 if homozygous in the alternate allele, 1 if heterozygous and 0 if homozygous in the reference or missing).

Requirement

Create a Rust program called *wink* (Wits PLINK), that takes a PLINK data set as input and performs one of three actions on them, depending on command line option used:

- computes the per-individual missingness (for each individual determine the proportion of genotypes that are missing).
- computes the per-SNP alternate allele frequency.
- takes a PLINK input file in SNP-order format (third byte of magic number is 1) into a PLINK file that is in individual-order format (and so third byte of the magic number in the transformed output file must be 0).

Here are examples of how the program could be run

`wink --missing arthritis`

Take a PLINK data set *arthritis.bed*, *arthritis.bim*, *arthritis.fam* and computes the per-individual missingness, storing the results in *arthritis.imiss*. The output file should have the missingness of each individual on a line by itself in the same order as the input fam file. Just store the numbers, rounded to four decimal points

`wink --aaf arthritis`

Take a PLINK data set *arthritis.bed*, *arthritis.bim*, *arthritis.fam* and computes the alternate allele frequency of each SNP, storing the results in *arthritis.freq*. The output file should have the AAF of each SNP on a line by itself in the same order as the input bim file. Just store the numbers, rounded to four decimal points

`wink --trans10 arthritis`

Take a PLINK data set *arthritis.bed*, *arthritis.bim*, *arthritis.fam* where the input bed file is in SNP-major format (third byte 1) and transposes the data to produce an individual-order bed file called *arthritis.tbed*, with the correct magic number.

`wink --trans01 arthritis`

Take a PLINK data set *arthritis.tbed*, *arthritis.bim*, *arthritis.fam* where the input bed file is in individual-major format (third byte 0) and transposes the data to produce a SNP-order bed file called *arthritis.check*, with the correct magic number.

The code that implements this should be exactly the same as `--trans10` The purpose of this and the naming conventions is to provide one convenient way for you to check correctness of the code. You can convert a data set from SNP-order to individual order and back. In this case the files *arthritis.bed* and *arthritis.check* should be exactly the same (use the md5sum program)

Criteria for judging

The criteria for judging are, in order:

- correctness;
- good coding style and good use of Rust;
- speed;
- memory use.

Restrictions

The goal here is to write sequential code that is as fast as possible. You should write code and design data structures that accomplishes this task. You can structure your code in a way that maximises the possibilities that the Rust compiler can exploit pipelining or even vector processing. However, you may not either explicitly or through the use crates introduce any parallelism – that will come later. The goal here is for you to revise your skills in producing very efficient code.

To avoid any doubt and put everyone on the same playing field, you may use any *std* library (except, no parallelism!) and the CLP crate. If you want to use any other crate, you must ask me and I will publish an open list of crates that are allowable.

There are, of course, existing Rust PLINK libraries on the internet and you can study them. However, I would strongly discourage you from basing your solution on them. Partly, this is because this is a learning exercise in Rust and I want you to understand the underlying details and bit representations. Second, the libraries are generally *much* more complex than needed (this is not a criticism of these libraries since they have a much more general purpose and generally support parallelism) but the goal here is to make the best solution for *these* particular problems not a general solution and I think you are going to take more time to reverse engineer existing code than implementing it yourself from first principles. For this particular exercise simple data structures are likely to work well. Third, I am aware and have studied several so I am familiar with the code and know what it looks like. Keep it simple.

Good programming practice

Your code should compile without errors or warnings. You may not use the *allow* attribute to suppress warnings. I strongly suggest using Rust unit tests.

- test data sets will be made available.

The use of the **unsafe** keyword is forbidden.

Submission

The Rust project should be named *wink* and placed in the home directory of your account on the Core Research cluster. The project must build successfully on the cluster. The performance of your code will be tested by running `cargo build --release` and then running the binary code `target/release/wink`.

In addition, you must submit on Ulwazi, two documents. The first document should make a standard plagiarism declaration, signed.

> I _____ (Student number: _____ am a student registered for _____ in the year _____ I hereby declare the following:
>
> - I am aware that plagiarism (the use of someone else's work without their permission and/or without acknowledging the original source) is wrong.
> - I confirm that ALL the work submitted for assessment for the above course is my own unaided work except where I have explicitly indicated otherwise.
> - I have followed the required conventions in referencing the thoughts and ideas of others.
> - I understand that the University of the Witwatersrand may take disciplinary action against me if there is a belief that this in not my own unaided work or that I have failed to acknowledge the source of the ideas or words in my writing.
>
> Signature: _____ Date: _____

This should be followed by a detailed list of any help you received including human help (from whom, what), books used, GitHub sites consulted and how used, etc, etc.. Remember, particularly at this level, students have an obligation that there are no misunderstandings.

The early submission date for this assignment (5% bonus) is 12:00 on Monday, 3 April. The submission date is 08:00 on 5 April. There is a 10% penalty for submitting after 08:00 and before 12:00. After this the project will not be marked.

The second document should be a maximum of 1 page long explaining your solution. The document itself will not be marked but you can use it to help the marker understand your code. Of course your code should be well commented.

The submission date timestamp will be the latest of the time of upload to Ulwazi and any file in your submission directory.

Marking

The marks will come from (a) the performance of your code, (b) the quality of your code and (c) an oral interview assessing your understanding and insight.