CPU Algorithm Design

Exercise 2 **Students:** Vishal Mangukiya, Konstantin Benz

Task 1.1.x

(1) You use `std::optional<T>` to represent an object of type `T` that may or may not be present. It is a wrapper around the type `T` that can be either empty or contain a value. This is useful when you want to indicate that a value might not be available without using pointers or special values (like `nullptr` or `-1`). For instance, a `find_user(id)` function can return `std::optional<User>-std::nullopt` if the user does not exist, otherwise the found `User` object.

(2) `std::variant` is a type-safe union that can hold exactly one of several types. A `std::tuple` is a fixed-size collection of the same types. Therefore a `std::variant` is more compact memory-wise than a `std::tuple`, because it only needs to store the size of the largest type, while a `std::tuple` needs to store the size of each type.

(3) `std::pair` and `std::complex` give their two components a fixed semantic meaning (first/second, real/imaginary) and always contain exactly two elements. In contrast, `std::tuple` and `std::array` are purely structural containers whose members are accessed by position and have no predefined interpretation.

(4) `std::pair` and `std::tuple` can store heterogeneous types (each element may have a different type). `std::array<T, N>` and `std::complex<T>` are homogeneous: every stored value has the same type `T` (and, for `std::complex`, there are always exactly two such values).

(5) `std::complex<T>` is a domain-specific numeric abstraction: beyond holding two values, it models the algebra of complex numbers and overloads arithmetic operators (‘+‘, ‘-‘, ‘*‘, ‘/‘, ‘abs‘, ‘arg‘, . . .). The other templates are generic containers and provide no mathematical behaviour on their own.

Task 1.2.x

(1) Documented code including test inside file `main-121.cpp`.

(2) Documented code including test inside file `main-122.cpp`.

(3) Documented code including test inside file `main-123.cpp`.