**Overview**

This project involves writing parts of code that one would need to implement a genetic algorithm. These parts are as follows:

* **All** project code is to be in a **single** #include file called **project.hpp**.
* The project.hpp #include file must have an **#include guard**.
* The project.hpp #include file can only have content that is required on this page.
* The definitions in the project.hpp #include file **must** be contained in the **nested namespaces uwindsor\_2023w::comp3400::project**.
  + ASIDE: Remember when #including other header files those are not contained in those namespaces --but everything else in that file needs to be inside those namespaces. (Yes, I intended to require the use of nested namespaces.)
* min() overloaded definitions defined in the uwindsor\_2023w::comp3400::project namespace
* a levenshtein() function defined in the uwindsor\_2023w::comp3400::project namespace
* a class char\_mutator definition defined in the uwindsor\_2023w::comp3400::project namespace
* a mutate() definition defined in the uwindsor\_2023w::comp3400::project namespace, and,
* a crossover() function definition defined in the uwindsor\_2023w::comp3400::project namespace.

There also is a provided file, utils.hpp, that must be used (and #included in your project.hpp). A link to this file appears later on this page.

You must implement project.hpp using what is required herein. That said, within such there is flexibility to implement things as you see fit.

You will need and should write your own .cpp files each with a main() function to test your code. Some example files have been provided as well (see links later on this page).

**Task**

**Starting project.hpp**

Start by:

* Creating a file called project.hpp.
* Adding an #include guard for project.hpp.
* Download utils.hpp (see link later on this page).
* Adding an #include "utils.hpp" to project.hpp.

Your will need to add other #include files to project.hpp as is needed below. Similarly, you will be adding code to project.hpp as required below.

**min() Overloaded Definitions**

As you already have learnt, std::min() exists in the C++ Standard Library. That version of min() only supports two arguments. You will write your own min() definitions to allow one or more arguments to be supported. These overloads must all be defined in the namespace uwindsor\_2023w::comp3400::project namespace in project.hpp.

The min() definitions must all have suitable defintiions using these prototypes:

template <typename T>  
constexpr T const& min(T const& a); // i.e., returns a  
  
template <typename T>  
constexpr T const& min(T const& a, T const& b); // i.e., like std::min()  
  
// definition of min() with three or more arguments...  
template <typename T, typename... Rest>  
requires (std::same\_as<T,Rest> && ...) // i.e., a [C++17](https://moodle.cs.uwindsor.ca/mod/glossary/showentry.php?eid=48&displayformat=dictionary) fold expression requiring all Rest... types to be the same as T  
constexpr T const& min(T const& a, T const& b, Rest const&... rest);

With each of these overloads min() must return the smallest value of all of the arguments passed as determined by the < (less than operator).

* Hint: For the variadic definition of min(), compare the first two arguments and then use "recursion" where there is one less argument passed to min. Such will eventually resolve to the single-argument min() overload, or, the double-argument min() overload. (Remember when using recursion the number of elements to be processed must decrease by at least one.)

When done, code such as:

std::cout << min(6,3,9) << '\n';

should output the smallest value of all arguments passed to it, e.g., 3 in this instance. (The above definitions require all types passed to be the same.)

**The levenshtein() Function Definition**

You are to write code to compute the Levenshtein edit distance between the two ranges a and b per the "Iterative with two matrix rows" code shown on this Wikipedia.org page <https://en.wikipedia.org/wiki/Levenshtein_distance#Iterative_with_two_matrix_rows> . Just in case it is edited/deleted, this code is quoted below:

function LevenshteinDistance(char s[0..m-1], char t[0..n-1]):

// create two work vectors of integer distances

declare int v0[n + 1]

declare int v1[n + 1]

// initialize v0 (the previous row of distances)

// this row is A[0][i]: edit distance from an empty s to t;

// that distance is the number of characters to append to s to make t.

for i from 0 to n:

v0[i] = i

for i from 0 to m - 1:

// calculate v1 (current row distances) from the previous row v0

// first element of v1 is A[i + 1][0]

// edit distance is delete (i + 1) chars from s to match empty t

v1[0] = i + 1

// use formula to fill in the rest of the row

for j from 0 to n - 1:

// calculating costs for A[i + 1][j + 1]

deletionCost := v0[j + 1] + 1

insertionCost := v1[j] + 1

if s[i] = t[j]:

substitutionCost := v0[j]

else:

substitutionCost := v0[j] + 1

v1[j + 1] := minimum(deletionCost, insertionCost, substitutionCost)

// copy v1 (current row) to v0 (previous row) for next iteration

// since data in v1 is always invalidated, a swap without copy could be more efficient

swap v0 with v1

// after the last swap, the results of v1 are now in v0

return v0[n]

Note the following about the wikipedia.org code:

* The code uses 0-based arrays --just like C, C++, and Java so this simplifies dealing with writing such in C++.
  + Remember when querying the size of a range, that the actual indices are 0 to size-1 and size is one larger than the highest index. (This is important when determining what size should v0 and v1 be declared as; determining if the for loop i and j values are correct; etc.)
* Declare v0 and v1 to be of type std::vector<std::size\_t>. (You can rename v0 and v1 to other names if you want.)
* Any indices involving items in ranges/std::vector must be of type std::size\_t.
* Tip: To obtain the size of a range in [C++20](https://moodle.cs.uwindsor.ca/mod/glossary/showentry.php?eid=49&displayformat=dictionary), use std::ranges::size(range\_variable).
* Tip: To obtain the begin iterator of a range use std::ranges::begin(range\_variable).
* Tip: To refer to a random-access index of a range, use the operator[]() member of its iterator just as you would with an array,.
  + ASIDE: Random-access iterators support a number of operations including indexing via operator[] --so if one has an iterator one can use (positive or negative) an index to reference items relative to that iterator.
* Tip: To initialize all elements of a std::vector object to a single value, use the appropriate constructor that will do that.
* Tip: Assuming there is at least one object in the std::vector object, to access the first element of a std::vector object one can simply call its .front() member function
* You may only call these C++ Standard Library functions to implement this function:
  + std::ranges::begin(), std::ranges::end(), std::ranges::size(), std::ranges::cbegin(), std::ranges::cend()
  + std::iota()
  + std::swap()
  + std::distance(), if needed
  + + you may use operations supported by obtained iterators.
* You may only use these data types to implement this function:
  + std::size\_t
  + std::vector<size\_t>
  + The type returned by std::ranges::begin().
  + The type returned by std::ranges::end().
  + The type returned by std::ranges::size().
  + The type returned by std::ranges::cbegin().
  + The type returned by std::ranges::cend().
* Tip: Iterate over the v0 and v1 vector's using std::size\_t indices, and, use some iterator variables with [] to access range indices.
  + Optional: Can i be an iterator to the first range? Yes: i is only used to index that range, and, it is only used to set the first value of v1 each time the loop starts. The latter can be computed using std::distance(ranges::begin(a),i)+1 or by creating another size\_t variable that counts how many times the outer for loop has iterated.
  + Optional: Can j also be an iterator to the second range? Yes. The issue is converting the use of indices properly over to iterators + ensuring that the values of the cost are computed as size\_t integer values. To change j to be an iterator, immediately before the j for loop, create two iterators starting at the beginning of their vectors: one for v0 and one for v1. Ensure these iterators along with j are incremented each time the j for loop iterates. Inside the j for loop, use [1] on the iterator to get the j\_index+1 value, \* with the iterator to get the current position's value, etc. Also since j is an iterator to the second range, one can now write \*j to access the current element. Not using integer indices for i and j results in one needing a couple of more iterator variables (i.e., for v0 and v1).
    - ASIDE: Is there much value in doing this? In this project with its requirements, it doesn't matter. More generally, yes it is worth doing if one wanted to relax the constraints that each range passed to the function is, say, a forward range which would have one would prefer having i and j as iterators to the ranges + a couple of iterator variables to v0 and v1. Why? Because these iterator variables move forward in a forward-only, sequential fashion so one can replace the use of iter[1] with \*std::next(iter) and everything would continue work even with forward and bidirectional ranges. The latter would result in an even more generic levenshtein() function definition: one could compute the Levenshtein distance between a std::set<T> (i.e., a balanced tree of T) and a std::list<T> (i.e., a list of T). This shows how one can start with code using arrays and work towards creating an increasingly more generic solution capable of handling many data structures --not just arrays! Cool, eh?!!! (NOTE: This ASIDE is not a project requirement --it is a FYI item.) :-)

The function prototype that must be used for this function is:

template <typename StringA, typename StringB>  
requires  
 std::ranges::sized\_range<StringA> &&  
 std::ranges::sized\_range<StringB> &&  
 std::ranges::random\_access\_range<StringA> &&  
 std::ranges::random\_access\_range<StringB> &&  
 std::same\_as<  
   std::ranges::range\_value\_t<StringA>,  
   std::ranges::range\_value\_t<StringB>  
 >  
std::size\_t levenshtein(StringA const& a, StringB const& b);

A test-levenshtein.cpp program has been provided (see link later on this page) that demonstrates testing some strings. Certainly, create some of your own tests as well.

**The class char\_mutator Definition**

The char\_mutator class must be defined as follows:

* char\_mutator must be a (non-template) class
* It must have three private variables:
  + a std::string variable that will hold "valid" characters,
  + a mutable std::uniform\_int\_distribution<std::size\_t> variable, and,
  + a mutable std::default\_random\_engine variable.
* It must have exactly one (public) operator() overload defined as follows:
  + The overload must return a char.
  + The overload must be a const member function.
  + The overload must be a member function template with its template parameter: typename... Args.
  + The overload must accept the arguments Args&&... --but don't define a variable name as arguments will always be ignored.
  + The overload must compute (and therefore return) the following:
    - First use the uniform distribution member and the range engine member variables to generate an index.
    - Second return the value at that index in the string member variable.
* Finally, char\_mutator must have a (public) default constructor as follows:
  + The three member variables must (only!) be initialized in the constructor member initialization list. (As will be (implicitly) apparent below, the string member variable needs to be constructed before the uniform dsitribution variable in this list.)
  + The string member variable construction is tricker: either use a lambda function that is immediately called, or, write a private member function that returns what needs to be passed to the string constructor. See below for the details on what needs to be passed to the string member variable's constructor.
  + The uniform distrbution member variable needs generate a value from 0 up to but not including value of the size() of the string variable.
    - i.e., the distribution is being used to generate an index of the string member variable.
  + The random engine member variable should be seeded with std::random\_device{}().
    - i.e., this will construct a std::random\_device variable and then immediately use its operator() member function to obtain a random integer which is then passed as the seed to the random engine constructor.
  + The default constructor's body MUST be empty.
    - i.e., you must initialize everything via constructors in the constructor member initialization list.

To compute the string returned and passed to the string member variable's constructor, first note the following:

* If you use a lambda function, write the lambda function and after the closing } (curly brace), immediately call it by writing ().
  + This is a clever, commonly used trick to compute something that one needs to pass to a constructor / initailize an object with.
  + For example, a const variable must be initialized when they are constructed. Sometimes one doesn't want to have all kinds of variable litter, etc. for code that only does that computation once, and, often one \*wants\* that variable to be const. A typical fix is to do the computational work needed inside a lambda function and return the value needed for the initialization by immediately calling the lambda function.
* If you are inside of a class, you can use a lambda function, or, you could also define a static private member function to do the same. The latter is useful when one needs to use that function in multiple locations. (A lambda function is really only useful if only needed in one location.)

You may choose to use a lambda function or a private static member function. Either way the function you write must compute the following (in order to return a string that is then passed to the string member's constructor):

* Declare a std::string variable. (This variable will returned at the end of the function and will be called "retval" below.)
* Write a for loop that iterates through all char values from 0 to std::numeric\_limits<char>::max().
  + Tip: If your counter variable is of type char, adding +1 to the maximum value is problematic. (Do remind yourself why this is appropriate and will work.) Avoid that problem and use a counter variable that is of type short. (Do remind yourself why this is appropriate and will work.)
* Inside the for loop if std::isalnum() or std::ispunct() or your counter variable is equal to a ' ' (space character), then append that character value to retval.
  + Tip: Since retval holds char values, calling a function to add an element means that element's type is char, i.e., so this means you can pass a short value to the char value (due to implicit conversions between integer types' rules). So if you were concerned that the counter variable is a short --that is a good concern but in this case it is not an issue since the value of the short variable is within the range of a char and thus there are no conversion issues.
* Return retval.

So in char\_mutator there are:

* three (3) private member variables,
* a default constructor,
* a static private member function (only if you chose that instead of a lambda), and,
* an operator() const member function overload.

Look at the test\_levenshtein.cpp program to see how this function is invoked and see Sample Program Runs to see a sample run of what it outputs.

ASIDE: A mutable class member allows that member to be changed even when a member function is const. Typically this is used for internal state variables such as variables used for caches or, as in this case, an internally used pseudo-random number generator and its distrbution (which have changing internal state that is not accessible to the public user of the class).

ASIDE: While Args&&... was used to accept any number of arguments in this class. One could have also written a member function non-template definition that accepted ... (i.e., the ellipsis operator and nothing else) in this case as well. What is the difference? Little except the compiler with the ellipsis operator definition would only generate one function. With the member template definition one function would be generated for each set of instance types passed to the operator() overload --so the latter would generate more compiled code --so if one wanted to simply ignore all arguments, consider using the ellipsis operator instead *and document that for other programmer's to know why you did that*. :-)

**The mutate() Function Definition**

The mutate function will mutate each of the individiual range's elements uniformly with probability rate per urbg. The mutation operator, m, is an invoacble  object that is also passed what the value to be mutated is. (This allows a caller to supply a mutation function that takes in to account the original value of the element when mutating.)

The mutate() function declaration/definition must have this prototype:

template <                                                                         
  std::ranges::range Individual,                                                   
  typename MutateOp,                                                               
  typename URBG                                                                    
>                                                                                  
requires   
 std::uniform\_random\_bit\_generator<std::remove\_cvref\_t<URBG>> &&  
 std::invocable<MutateOp,std::ranges::range\_value\_t<Individual>>           
void mutate(                                                                       
  Individual& individual,                                                          
  double const rate,                                                               
  MutateOp&& m,                                                                    
  URBG&& urbg                                                                      
);

The URBG (uniform random bit generator) parameter:

* All C++ Standard Library random engines are URBGs.
* The same UBRG objects that can be passed to std::sample() can be passed to this argument.
* [cppreference.com](https://moodle.cs.uwindsor.ca/mod/glossary/showentry.php?eid=52&displayformat=dictionary) describes URBGs here: <https://en.cppreference.com/w/cpp/named_req/UniformRandomBitGenerator>
* i.e., simply pass in a random engine to this parameter when testing your code.

The mutate function must be implemented as follows:

* Declare a std::uniform\_real\_distribution<double> variable so that it generates values between 0.0 and 1.0.
* Invoke std::ranges::for\_each() on individual.
* The lambda function passed to the for\_each function must be as follows:
  + Capture any needed variables by reference.
  + The lambda function parameter should be a non-const lvalue reference. Why? The element might be modified (see below). This argument is referred to as "arg" below.
  + Obtain a random number using urbg with the uniform distribution variable you declared.
  + If the random value is less than rate, then **assign arg to m(arg)**, i.e., this will change/mutate that element.

Look at the test\_mutate.cpp program to see how this function is invoked and see Sample Program Runs to see a sample run of what it outputs.

**The crossover() Function Definition**

Unlike the above, this function is a longer function and trickier to write. Think and plan what you conceptually aim to do before writing code --this will make writing, testing, debugging, etc. this function much easier.

The purpose of the crossover function is to perform a k-point crossover (e.g., per <https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)#Two-point_and_k-point_crossover>). Crossover one of the operations in a [Generic Algorithm](https://en.wikipedia.org/wiki/Genetic_algorithm) which is a problem solving heuristic technique that borrows inspired ideas from meiosis (i.e., sexual reproduction). In sexual reproduction, each child individual has genetic material from both parent indidividuals (approximately 50% from each parent). Crossover occurs in meiosis with one strand of each parent's DNA being paired up with each other. Portions are then swapped in/out from each parent along the DNA strands. Each crossover point changes which parent's DNA is copied to the "child" DNA strand. This function treats each parent's single strand of "DNA" as a [C++20](https://moodle.cs.uwindsor.ca/mod/glossary/showentry.php?eid=49&displayformat=dictionary) range whose size is can be determined in constant time.

The function prototype of the crossover() function template must be:

template <typename URBG1, typename URBG2, typename Individual>  
requires  
 std::uniform\_random\_bit\_generator<std::remove\_cvref\_t<URBG1>> &&  
 std::uniform\_random\_bit\_generator<std::remove\_cvref\_t<URBG2>> &&  
 std::ranges::forward\_range<Individual> &&  
  std::ranges::sized\_range<Individual> &&  
 smart\_insertable<Individual>  
auto crossover(  
 std::size\_t const ncrossover\_points,  
 URBG1&& urbg\_starting\_parent,  
 URBG2&& urbg\_crossover\_points,  
 Individual const& parent1,  
 Individual const& parent2  
) -> std::remove\_cvref\_t<Individual>;

This function requires two URBG (uniform random bit generators) since one is used to determine which parent is used first to start copying from and the other is used to determine crossover point indices (of which ncrossover\_points are generated). By using two independent random sequences, this function avoids causing a sequence to not be random had only one URBG been used. The arguments parent1 and parent2 must be of the same type and the return type is a child whose type is the same as the parent type except with const, volatile, and references removed from the parent type (if such was present).

ASIDE: std::remove\_cvref\_t<Type> is defined in <type\_traits>. Look it up and see what it does. Take a look of other definitions in that header file. While not the same as other languages, this, along with (sometimes using) [C++20](https://moodle.cs.uwindsor.ca/mod/glossary/showentry.php?eid=49&displayformat=dictionary) concepts, is closest in C++ to what other languages call "reflection".

Look at the test\_crossover.cpp program to see how this function is invoked and see Sample Program Runs to see a sample run of what it outputs. (Also notice that if one parent range is longer than the other parent's range the last segment copies everything to the end of the (appropriate) parent range. This simplifies handling ranges that have different lengths.)

ASIDE: Do look at the code and sample program run of test\_crossover.cpp output before continuing. The test\_crossover.cpp program uses strings to make it easy to tell which strings belong to which parent --which, in turn, makes it much easier to see and understand what the crossover() function does.

The crossover function requires the following::

* NOTE: The URBG1 and URBG2 parameters are uniform random bit generators.
  + The urbg\_starting\_parent argument is used to determine which parent to start copying from.
    - This URBG will be used with std::bernoulli\_distrbution.
  + The urbg\_crossover\_points argument is used to determine the crossover indices to switch which parent is copied from for that "chunk".
    - This URBG will be used with std::uniform\_int\_distribution.
* NOTE: The parent1 and parent2 parameters are const forward ranges with constant-time size() support.
  + ASIDE: Each parent range is considered to be that parent's "DNA".
  + ASIDE: The test\_crossover.cpp program intentionally uses parent ranges that have different lengths so you can easily see how such is handled. (Normally with genetic algorithms one ensures both parents have the same length.)
* You may only use these C++ Standard Library types and function calls in this functions code:
  + std::adjacent\_difference()
  + std::advance()
  + std::back\_inserter()
  + std::bernoulli\_distribution
  + std::ranges::cbegin()
  + std::ranges::cend()
  + std::ranges::copy()
  + std::copy\_n()
  + std::iota()
  + std::remove\_cvref\_t<Individual>
    - ASIDE: You will need to use this to declare a variable to hold the return value. If you wrote Individual variable\_name; you might discover that might not work if Individual had const or reference decorations. Removing such eliminates the issue, e.g., write std::remove\_cvref\_t<Individual> variable\_name; to declare the variable that will be used to hold the computed result that need to be returned from the function.
  + std::ranges::sample()
    - ASIDE: Note [C++20](https://moodle.cs.uwindsor.ca/mod/glossary/showentry.php?eid=49&displayformat=dictionary) ranges implement most of the <algorithm> and <numeric> functionss in std::ranges (defined in <ranges>). Typically the difference is that the std::ranges namespace functions allow one to pass in a single-argument range variable instead of two arguments that are iterators. std::ranges::sample() is the same as std::sample() except the very first argument is a range --not two iterators.
  + std::ranges::size()
  + reserve\_or\_noop() in the provided utils.hpp (see link later on this page)
    - ASIDE: Should the containers/ranges you are using have a reserve() function call to reserve sufficient RAM/entries for what is being added, calling this function will perform such --or it will do nothing at all. See the code in utils.hpp.
  + std::size\_t
  + smart\_inserter() in the provided utils.hpp (see link later on this page)
    - ASIDE: Since the Individual type is a template parameter, you don't know exactly what type it is, e.g., it could be std::set<char>, std::string, or something else. Consequently you don't know how you will insert/append data to such.
  + std::vector<size\_t>
  + + the Standard Library member functions, types, etc. directly associated with / returned by the above including containers and iterators.

The crossover() function must implement a k-point genetic algorithm crossover where:

* k is >= zero (0),
* the k crossover points are selected from positions found in both parent's ranges, and,
  + ASIDE: If one parent is longer than the other, the extra amount is ignored in terms of performing crossovers.
* only one child is returned (some implementations of the crossover function return two children).

Some may find this function more of a challenge to implement. Here are two strategies:

* Strategy A: Start by figuring out how to write this function by writing a solution hard-coding it to work with std::string objects (i.e., for parent1 and parent2) and use normal std::string iterators and indicies. Once one figures out how to make that work, that could then be used as a basis to write this project's crossover() function template. (The logic/calls will be the same/equivalent and such should work fairly easily since the testing/debugging was already done with hard-coded std::string version.)
* Strategy B: One could write this code using the required prototype above --but don't write the whole function before starting to compile and test things. Instead write small bits and ensure those bits work first then add more.

It is very important to think about and sketch out (e.g., using a notepad, etc.) how you will solve implementing the crossover. First understand what performing the crossover is, e.g., look at the test\_crossover.cpp program and its sample output below. Then think about how you would solve this. Remember each crossover point is BETWEEN two range elements --but one uses iterators/integes so one cannot store "0.5" amounts to be "between".

ASIDE: Think of parent1 and parent2 being side-by-side strings. Each crossover is a twisting of the two strands 180 degrees --so the strang on "left-hand side" flips to be on the "right-hand side" after a crossover and similarly for the "right-hand side" strand. Between each crossover point are data (i.e., some parent1/parent2 elements). If this is not fully appreciated and coded properly, your code will likely be off by one with issues in terms of what it copies.

To help you write this function, the following outlines and/or gives hints and tips as to how to write this function. That said, what follows is not a reason to not think about, etc. how to write the code for this function.

* You will need to know the minimum size of the two ranges. Use min() that you wrote earlier along with std::ranges::size, to determine the minimim size of the two parents. Store this in a read-only variable.
  + ASIDE: This variable will be referred to as "psize\_truncated" below.
* You will need to use the std::bernoulli\_distribution. Declare a std::bernoulli\_distribution variable with probability 0.5.
* Since there are two parents, you can represent such with any exact type that supports at least two distinct values, e.g., bool. Declare a variable that identiifies which parent range is currently being copied from and initialize it to the std::bernoulli\_distribution variable being applied to urbg\_starting\_parent. (This will generate a random true/false value, i.e., one value will be set to mean parent1 and the other will be set to mean parent2.)
  + ASIDE: This two-state variable will be referred to as "which\_parent" below.
* To make it easier to write subsequent code, write an if statement such that if any of ncrossover\_points or parent1's range size or parent2's range size are 0, then return the parent argument as-is per the which\_parent variable's value.
  + ASIDE: Thus, after this if statement, one knows that neither parent's range size nor ncrossover\_points are all at least one.
  + ASIDE: Basically, if there are no crossover points, then one is randomly returning one of the parents.
* Realize the number of crossover points, k, you need is stored in the "ncrossover\_points" function argument variable.
* Declare a std::vector<size\_t> variable to hold the indices of each crossover point.
  + Construct this vector with psize\_truncated-1 elements. (This needs to be done in order to use std::iota().)
  + This variable will be referred to as "crossover\_indices" below.
* Call std::iota() over all elements in crossover\_indices starting at 1 --not 0.
  + ASIDE: Think about why you might want to do this starting at 1 and not 0.
  + ASIDE: Also why have an array of indices counting upwards? This allows one to use std::ranges::sample() to uniformly sample (without replacement) the crossover points. Cool eh?!!! :-)
* Declare a std::vector<size\_t> to hold the crossover offset values.
  + This variable will be referred to as "crossover\_offsets" below.
  + Until std::adjacent\_difference() is called below, the elements are really indices --not offsets. After std::adjacent\_difference() is called on this vector all of the values will be relative offsets. Such will be needed to easily copy alternating "chunks" from each parent.
* Call reserve\_or\_noop() on crossover\_offsets with ncrossover\_points.
  + ASIDE: Why? This will help improve the code efficiency as crossover\_offsets grows in size.
* Call std::ranges::sample() on the vector crossover\_indices, using std::back\_inserter() to append elements to crossover\_offsets, sampling ncrossover\_points with the urbg\_crossover\_points UBRG.
  + ASIDE: Remember a C++ Standard Library container, e.g., std::vector, is a range --so it can be passed to std::ranges::sample().
* Now with crossover\_offsets populated with some crossover points, call std::adjacent\_difference() to compute the adjacent differences of each index. This will make the crossover\_offsets vector have relative offsets starting the the beginning. This will allow one to use iterators to track iterator positions in each parent when copying data.
  + ASIDE: If you did not realize it yet, this is why you want std::iota() to start at 1. If you start at 0, then the first "chunk" to copy might be of size zero. Remember earlier you determined which\_parent to start copying from --and you want to copy at least 1 element from a parent each time.
  + ASIDE: The reason crossover\_indices has psize\_truncated-1 elements instead of psize\_truncated elements is that you want to copy all remaining elements from the parent in the last chunk. Since the parents might not be the same size, this allows you to stop looping just before the last chunk is copied so you can perform the copy of the last chunk copying all remaining elements.

Okay if you're at this point, all that needs to be done now is to copy alternating chunks from each parent in to a container that is returned from the function. Guidance on how this can be achieved:

* Declare the return value variable of type Individual with any const, volatile, and reference removed from the Individual type.
  + ASIDE: This variable will be called "retval" below.
* Call reserve\_or\_noop() on retval with the result of std::max() of std::ranges::size() of parent1 and std::ranges::size() of parent2.
  + ASIDE: This std::max() call is max() in the C++ Standard Library. (Earlier the definition of min() used was the one you wrote.)
  + ASIDE: Why do this? Efficiency. Also one does not know in this function the type of Individual which could be a std::vector, std::string, std::list, etc.
* Call smart\_inserter passing retval storing the return value in a variable.
  + ASIDE: This variable will be referred to as "out" below.
  + ASIDE: Look at utils.hpp to see what smart\_inserter() does.
* Declare a variable whose initial value is std::ranges::cbegin(parent1);
  + ASIDE: This variable will be referred to as "p1pos" below.
* Declare a variable whose initial value is std::ranges::cbegin(parent2);
  + ASIDE: This variable will be referred to as "p2pos" below.
* Write a for loop that iterates over all elements in crossover\_offsets.
  + Within the for loop, each element will be referred to as "offset".
  + If which\_parent is set to the parent1 value, then call std::copy\_n() passing in p1pos, offset, and out. Assign the return value of std::copy\_n() to out.
  + If which\_parent is set to the parent2 value, then do the same as for parent1 except with parent2.
  + Call std::advance() on p1pos adjusting it by offset.
  + Call std::advance() on p2pos adjusting it by offset.
  + Change which parent to be the value representing the other parent.
    - Tip: If you used a bool for which\_parent, you can "which\_parent = !which\_parent;" to toggle its value between the two states. Using a bool avoids having to have more if statements, etc. If also makes it easy to test for "which\_parent" in if statements since one can write lines of code like this: "if (which\_parent) /\* parent1 code \*/; else /\* parent2 code \*/;"
* After the for loop, if which\_parent is set to the value for parent1, then call std::ranges::copy() on the range [p1pos,std::ranges::cend(parent1)) passing in out. Otherwise do the same but for p2pos and parent2.
  + ASIDE: Notice this copies the last chunk to the end of the appropriate parent range.
  + ASIDE: The reason for using std::ranges::copy() is because std::ranges::end(), etc. can return sentinel values in general, i.e., it might not return an iterator which std::copy() requires. (Know that std::ranges::begin(), etc. always return iterators).
* Return retval.

**Sample Program Runs**

This project has some provided files (see the links to such below). You are to write the code in project.hpp as outlined earlier on this page. That said, sample runs of the provided test\_\*.cpp files are provided below (run against a correct project.hpp) so you can see what correct output looks like. (Remember these outputs make use of random number generators seeded with std::random\_device.)

$ ./test\_levenshtein.exe  
11111111111  
$ ./test\_mutate.exe  
"To be or not to be."  
"To>bt or not ?^ Te."  
"To be or nPtDto bef"  
"Zo Ve o& not7to K`."  
"To be or not to>$e."  
"To be ;0 not to be."  
"TK b` oB lot to be."  
"do T:@^r @ot t6\\be."  
"Tj 8emvr :o| t< be~"  
"^o b-uoY bot toUTe."  
"W@ be or 3oteto b;."  
$ ./test\_crossover.exe  
p1: "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  
p2: "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"  
  
0: "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  
0: "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"  
0: "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"  
0: "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"  
0: "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"  
0: "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"  
0: "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  
0: "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  
  
1: "XXXXXXXXXXXXXXXXXXXX\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  
1: "XXXXXXXXX\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  
1: "\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"  
1: "XXXXXXXX\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  
1: "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXXXXXXX"  
1: "XX\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  
1: "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXXXXXXXXX"  
1: "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXXXXXXXXXXX"  
  
2: "\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXXX\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  
2: "XXXXXX\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXXXX"  
2: "\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXX\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  
2: "XXXXXXXXXXXXXXXXXXXXXXXXXXXX\_XXXXXXXXXXXXXXXXX"  
2: "XX\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXX"  
2: "XXXXX\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXXXXXXXXXXXXXXX"  
2: "\_\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXX\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  
2: "\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXXXXXX\_\_\_\_\_\_\_\_\_\_\_"  
  
3: "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXX\_\_\_\_\_\_\_\_XXXXXXXXXX"  
3: "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXXXXX\_\_\_\_\_\_XXXXXXX"  
3: "\_XXXXXXXXXXXXXXXXX\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXX"  
3: "XXXXXXXXXXXXXXXX\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXX\_"  
3: "XXXXX\_\_\_\_\_\_XXXXXXXXXXXXXX\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  
3: "XXXXXXXXXXXXXXXXXXXXXXX\_\_\_\_\_\_XXX\_\_\_\_\_\_\_\_"  
3: "\_\_\_\_\_\_XXXXXXXX\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_XXXXXXXXXX"  
3: "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_X\_\_\_\_\_\_\_XXXXXXXXXXXXXXXXX"  
  
0: "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"  
38: "X\_X\_X\_X\_X\_XX\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_XXXXXXX"  
18: "\_\_XXXXX\_XXXX\_XXXXXX\_X\_X\_\_\_XXX\_X\_\_\_\_X\_X\_\_"  
21: "X\_XX\_X\_\_XXXX\_\_X\_XXX\_XX\_\_\_X\_X\_\_X\_\_\_\_XXX\_\_"  
1: "\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"  
23: "\_XXX\_\_XXX\_\_X\_\_X\_XXX\_\_XX\_\_XXXX\_XX\_X\_X\_X\_XXXXXXX"  
26: "X\_\_\_X\_\_X\_X\_XX\_\_\_X\_X\_\_X\_XXX\_XX\_X\_\_XX\_X\_\_XXXXXXX"  
36: "X\_X\_X\_X\_X\_X\_\_X\_X\_X\_X\_\_X\_X\_X\_X\_X\_X\_X\_X\_XXXXXXXX"  
19: "\_X\_\_\_\_\_XXX\_X\_XX\_X\_\_\_XX\_\_\_XX\_XXXX\_\_\_\_XX\_XXXXXXX"  
38: "X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_XX\_X\_X\_X\_X\_X\_XXXXXXX"  
25: "XXX\_X\_X\_X\_\_X\_XX\_XXX\_X\_XX\_XX\_\_\_X\_X\_XXXX\_\_"  
2: "\_\_\_\_\_\_\_\_\_\_XXXXXXXXXXXXXXXXXXXXXXXXX\_\_\_\_\_"  
23: "\_X\_\_X\_\_X\_\_\_\_\_XX\_X\_\_\_XX\_\_X\_\_XXX\_X\_\_X\_XX\_XXXXXXX"  
8: "\_\_\_\_\_\_\_\_XXXX\_\_\_\_\_X\_\_\_\_\_\_\_\_\_\_\_X\_X\_\_\_\_\_\_\_\_"  
10: "XXXXXXX\_\_XXXXX\_\_\_X\_\_\_\_\_\_\_\_\_\_\_\_XXXX\_\_\_X\_XXXXXXX"  
11: "\_\_\_XXXXX\_\_\_X\_\_\_\_X\_XXXXXXX\_XX\_\_\_\_\_\_\_XXXXXXXXXXX"  
35: "X\_X\_X\_X\_X\_X\_X\_\_XX\_X\_X\_X\_X\_\_X\_X\_\_X\_X\_X\_X\_"  
23: "\_\_XX\_X\_XXX\_XXX\_X\_\_X\_XXXX\_\_X\_XX\_XX\_XXXX\_XXXXXXX"  
16: "\_\_XXXXXX\_\_\_\_\_X\_X\_\_\_XX\_\_\_\_X\_\_XXXX\_XXX\_X\_\_"  
39: "X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_X\_"

When compiling your project you may want to use these compiler options with GCC:

-std=[c++20](https://moodle.cs.uwindsor.ca/mod/glossary/showentry.php?eid=49&displayformat=dictionary) -Wall -Wextra -Werror -Wold-style-cast -fconcepts-diagnostics-depth=10 -g3 -fsanitize=address

The -fconcepts-diagnostics-depth=10 option tells GCC to output more error information on concept violations. (If you don't write this GCC will probably tell you to use such if you want to see more information.)

The -g3 options turns on debugging information generation. Besides being able to use a debugger like GDB, this is useful with the next option: -fsanitize=address. The latter option turns on run-time checks that help identify incorrect uses of unallocated RAM, etc. If there is an issue, the program will abort with dump of information --included in that are the line numbers involved at the time the memory problem was noticed. For example, if your code tries to copy data past the end of a parent range, this sanitize check will likely be triggered. Such would allow you to know there is indeed a real memory problem, and, to know the line numbers in your source code that triggered the issue. If this -f option is not used, then there will be no output and the program may or may not segfault.

These are the provided files

This is utils.hpp

#ifndef uwindsor\_2023w\_comp3400\_project\_utils\_hpp\_  
#define uwindsor\_2023w\_comp3400\_project\_utils\_hpp\_  
  
*//=============================================================================*#include <concepts>  
#include <iterator>  
#include <type\_traits>  
  
*//=============================================================================  
  
namespace* uwindsor\_2023w {  
*namespace* comp3400 {  
*namespace* project {  
  
*//=============================================================================  
  
template* <*typename T*, *typename Int*>  
*concept* reservable =  
 *requires* (*T t*, *Int n*)  
 {  
 { *t*.reserve(*n*) };  
 }  
;  
  
*template* <*typename Container*, *typename Int*>  
*constexpr void* reserve\_or\_noop(*Container*& *c*, *Int const*& *n*)  
{  
 *if constexpr*(reservable<*Container*,*Int*>)  
 *c*.reserve(*n*);  
}  
  
*//=============================================================================  
  
template* <*typename Container*>  
*concept* back\_insertable =  
 std::ranges::range<*Container*> &&  
 *requires* (*Container c*)  
 {  
 { std::back\_inserter(*c*) } ->   
 std::output\_iterator<std::ranges::*range\_value\_t*<*Container*>>;  
 *// NOTE: The std::output\_iterator<A,B> concept, to be true, needs the A   
 // parameter be an output iterator type and the B parameter to be   
 // a suitable value type. Using std::output\_iteator<T> here  
 // works because the compiler takes the type of what is returned   
 // from the expression inside { } and passes it as the first  
 // parameter of std::output\_iterator.* }  
;  
  
*template* <*typename Container*>  
*concept* insertable =  
 std::ranges::range<*Container*> &&  
 *requires* (*Container c*)  
 {  
 { std::inserter(*c*) } ->   
 std::output\_iterator<std::ranges::*range\_value\_t*<*Container*>>;  
 }  
;  
  
*template* <*typename Container*>  
*concept* smart\_insertable = back\_insertable<*Container*> || insertable<*Container*>;  
  
*template* <smart\_insertable *Container*>  
*constexpr auto* smart\_inserter(*Container*& *c*)  
{  
 *if constexpr*(back\_insertable<*Container*>)  
 *return* std::back\_inserter(*c*);  
 *else  
 return* std::inserter(*c*, end(*c*));  
}  
  
*//=============================================================================*} *// namespace project*} *// namespace comp3400*} *// namespace uwindsor\_2023w  
  
//=============================================================================*#endif *// #ifndef uwindsor\_2023w\_comp3400\_project\_utils\_hpp\_*

This is test\_crossover.cpp

*//=============================================================================*#include <iomanip>  
#include <iostream>  
#include <random>  
#include <string>  
#include "project.hpp"  
  
*//=============================================================================  
  
int* main()  
{  
 *using namespace* std;  
 *using* uwindsor\_2023w::comp3400::project::min;  
 *using* uwindsor\_2023w::comp3400::project::crossover;  
  
 *string const*& parent1{ "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" };  
 *string const*& parent2{ "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" };  
 cout   
 **<<** "p1:\t" **<<** quoted(parent1) **<<** '\n'  
 **<<** "p2:\t" **<<** quoted(parent2) **<<** "\n\n"  
 ;  
  
 random\_device rd;  
  
 *// Declare random engines for crossover()...  
 default\_random\_engine* which\_parent\_re{rd()};  
 *default\_random\_engine* crossover\_points\_re{rd()};  
   
 *for* (*int* i{}; i != 4; ++i)  
 {  
 *for* (*int* repeat{}; repeat != 8; ++repeat)  
 cout   
 **<<** i **<<** ":\t"   
 << quoted(crossover(i, which\_parent\_re, crossover\_points\_re,   
 parent1, parent2))   
 << '\n'  
 ;  
 cout **<<** '\n';  
 }  
  
 *// set up random engine and distribution for the number of crossover  
 // points...  
 default\_random\_engine* num\_crossovers\_re{rd()};  
 uniform\_int\_distribution<*size\_t*> num\_crossovers\_dist(  
 0, min(parent1.size(), parent2.size())  
 );  
  
 *for* (*int* i{}; i != 20; ++i)  
 {  
 *auto const* n = num\_crossovers\_dist(num\_crossovers\_re);  
 cout   
 **<<** n **<<** ":\t"   
 << quoted(crossover(n, which\_parent\_re, crossover\_points\_re,   
 parent1, parent2))   
 << '\n'  
 ;  
 }  
}  
  
*//=============================================================================*

This is test\_mutate.cpp

*//=============================================================================*#include <iomanip>  
#include <iostream>  
#include <random>  
#include <string>  
#include "project.hpp"  
  
*//=============================================================================  
  
int* main()  
{  
 *using* uwindsor\_2023w::comp3400::project::mutate;  
 *using* uwindsor\_2023w::comp3400::project::char\_mutator;  
  
 std::string *const* str{ "To be or not to be." };  
 std::cout << std::quoted(str) << '\n';  
  
 char\_mutator m;  
 std::default\_random\_engine re{std::random\_device{}()};  
 *for* (*int* i{}; i != 10; ++i)  
 {  
 *auto* str2 = str;  
 mutate(str2, 0.25, m, re);  
 std::cout << std::quoted(str2) << '\n';  
 }  
}  
  
*//=============================================================================*

This is test\_levenshtein.cpp

*//=============================================================================*#include <array>  
#include <iostream>  
#include <string>  
#include <vector>  
#include "project.hpp"  
  
*//=============================================================================  
  
int* main()  
{  
 *using namespace* std;  
 *using namespace* std::literals;  
 *using* uwindsor\_2023w::comp3400::project::levenshtein;  
  
 cout   
 << (levenshtein("kitten", "sitting") == 3)   
 << (levenshtein("Saturday", "Sunday") == 3)  
 << (levenshtein("thou shalt not", "you should not") == 5)  
 << (levenshtein("","") == 0)  
 << (levenshtein(""s,""s) == 0)  
 << (levenshtein(""s,vector<*char*>{}) == 0)  
 *// NOTE: A C-style string literal implicitly has and includes a '\0'  
 // character at the end of the string which is included in its   
 // "length" as a literal value. This means the next line returns an  
 // edit distance of 1 instead of zero for an "empty" string.* << (levenshtein(""s,"") == 1)  
 << (levenshtein(string{"house"},"mouse"s) == 1)  
 << (levenshtein(vector{'c','a','r'}, array{'b','a','t'}) == 2)  
 << (levenshtein(wstring{*L*"αβδε"}, *L*"αβ\_δε") == 2)  
 << (levenshtein(vector{'V','s','a','u','c','e'}, "apple sauce"s) == 6)  
 << '\n'  
 ;  
}  
  
*//=============================================================================*