PROJECTS FOR SCIENTIFIC PROGRAMMING IN C++ AND OTHER LANGUAGES

VICTOR EIJKHOUT

2020

Contents

PART I

SIMPLE PROJECTS

Chapter 1

Prime numbers

In this chapter you will do a number of exercises regarding prime numbers that build on each other. Each section lists the required prerequisites. Conversely, the exercises here are also referenced from the earlier chapters.

1.1 Arithmetic

Exercise 1.1. Read two numbers and print out their modulus. The modulus operator is `x%y`.

- Can you also compute the modulus without the operator?
- What do you get for negative inputs, in both cases?
- Assign all your results to a variable before outputting them.

1.2 Conditionals

Exercise 1.2. Read two numbers and print a message stating whether the second is as divisor of the first:

Code:

```
1  int number,divisor;
2  bool is_a_divisor;
3    /* ... */
4  if (
5    /* ... */
6      ) {
7    cout << "Indeed, " << divisor
8         << " is a divisor of "
9         << number << '\n';
10 } else {
11   cout << "No, " << divisor
12        << " is not a divisor of "
13        << number << '\n';
14 }
```

Output

[primes] division:

```
( echo 6 ; echo 2 ) |
    divisiontest
Enter a number:
Enter a trial divisor:
Indeed, 2 is a divisor of 6

( echo 9 ; echo 2 ) |
    divisiontest
Enter a number:
Enter a trial divisor:
No, 2 is not a divisor of 9
```

1.3 Looping

Exercise 1.3. Read an integer and set a boolean variable to determine whether it is prime by testing for the smaller numbers if they divide that number.

Print a final message

```
Your number is prime
```

or

```
Your number is not prime: it is divisible by ....
```

where you report just one found factor.

Printing a message to the screen is hardly ever the point of a serious program. In the previous exercise, let's therefore assume that the fact of primeness (or non-primeness) of a number will be used in the rest of the program. So you want to store this conclusion.

Exercise 1.4. Rewrite the previous exercise with a boolean variable to represent the primeness of the input number.

Exercise 1.5. Read in an integer r. If it is prime, print a message saying so. If it is not prime, find integers $p \leq q$ so that $r = p \cdot q$ and so that p and q are as close together as possible. For instance, for $r = 30$ you should print out `5,6`, rather than `3,10`. You are allowed to use the function `sqrt`.

1.4 Functions

chapter]ch:function

Above you wrote several lines of code to test whether a number was prime. Now we'll turn this code into a function.

Exercise 1.6. Write a function `is_prime` that has an integer parameter, and returns a boolean corresponding to whether the parameter was prime.

```
int main() {
  bool isprime;
  isprime = is_prime(13);
```

Write a main program that reads the number in, and prints the value of the boolean. (How is the boolean rendered? See section 12.1.2 (textbook).)

Does your function have one or two `return` statements? Can you imagine what the other possibility looks like? Do you have an argument for or against it?

1.5 While loops

Exercise 1.7. Take your prime number testing function `is_prime`, and use it to write a program that prints multiple primes:

- Read an integer `how_many` from the input, indicating how many (successive) prime numbers should be printed.
- Print that many successive primes, each on a separate line.
- (Hint: keep a variable `number_of_primes_found` that is increased whenever a new prime is found.)

1.6 Classes and objects

Exercise 1.8. Write a class `primegenerator` that contains:

- Methods `number_of_primes_found` and `nextprime`;
- Also write a function `isprime` that does not need to be in the class.

Your main program should look as follows:

```
cin >> nprimes;
primegenerator sequence;
while (sequence.number_of_primes_found()<nprimes) {
  int number = sequence.nextprime();
  cout << "Number " << number << " is prime" << '\n';
}
```

In the previous exercise you defined the `primegenerator` class, and you made one object of that class:

```
primegenerator sequence;
```

But you can make multiple generators, that all have their own internal data and are therefore independent of each other.

Exercise 1.9. The *Goldbach conjecture* says that every even number, from 4 on, is the sum of two primes $p + q$. Write a program to test this for the even numbers up to a bound that you read in. Use the `primegenerator` class you developed in exercise 45.8 (textbook).

This is a great exercise for a top-down approach!

1. Make an outer loop over the even numbers e.
2. For each e, generate all primes p.
3. From $p + q = e$, it follows that $q = e - p$ is prime: test if that q is prime.

For each even number `e` then print `e`, `p`, `q`, for instance:

```
The number 10 is 3+7
```

If multiple possibilities exist, only print the first one you find.

An interesting corollary of the Goldbach conjecture is that each prime (start at 5) is equidistant between two other primes.

The *Goldbach conjecture* says that every even number $2n$ (starting at 4), is the sum of two primes $p+q$:

$$2n = p + q.$$

Equivalently, every number n is equidistant from two primes:

$$n = \frac{p+q}{2} \qquad \text{or} \qquad q - n = n - p.$$

In particular this holds for each prime number:

$$\forall_{r \text{ prime}} \exists_{p,q \text{ prime}} : r = (p+q)/2 \text{ is prime.}$$

Exercise 1.10.

Write a program that tests this. You need at least one loop that tests all primes r; for each r you then need to find the primes p, q that are equidistant to it. Do you use two generators for this, or is one enough? Do you need three, for p, q, r?

For each r value, when the program finds the p, q values, print the p, q, r triple and move on to the next r.

1.6.1 Exceptions

Exercise 1.11. Revisit the prime generator class (exercise 45.8 (textbook)) and let it throw an exception once the candidate number is too large. (You can hardwire this maximum, or use a limit; section 24.2 (textbook).)

Code:

```
try {
  do {
    auto cur = primes.nextprime();
    cout << cur << '\n';
  } while (true);
} catch ( string s ) {
  cout << s << '\n';
}
```

Output
[primes] genx:

```
9931
9941
9949
9967
9973
Reached max int
```

1.6.2 Prime number decomposition

Design a class `Integer` which stores its value as its prime number decomposition. For instance,

$$180 = 2^2 \cdot 3^3 \cdot 5 \qquad \Rightarrow \qquad \texttt{[2:2, 3:2, 5:1]}$$

You can implement this decomposition itself as a *vector*, (the i-th location stores the exponent of the i-th prime) but let's use a `map` instead.

Exercise 1.12. Write a constructor of an `Integer` from an `int`, and methods `as_int` / `as_string` that convert the decomposition back to something classical. Start by assuming that each prime factor appears only once.

Code:

```
Integer i2(2);
cout << i2.as_string() << ": "
     << i2.as_int() << '\n';

Integer i6(6);
cout << i6.as_string() << ": "
     << i6.as_int() << '\n';
```

Output
[primes] decomposition26:

```
2^1 : 2
2^1 3^1 : 6
```

Exercise 1.13. Extend the previous exercise to having multiplicity > 1 for the prime factors.

Code:

```
Integer i180(180);
cout << i180.as_string() << ": "
     << i180.as_int() << '\n';
```

Output
[primes] decomposition180:

```
2^2 3^2 5^1 : 180
```

Implement addition and multiplication for `Integer`s.

Implement a class `Rational` for rational numbers, which are implemented as two `Integer` objects. This class should have methods for addition and multiplication. Write these through operator overloading if you've learned this.

Make sure you always divide out common factors in the numerator and denominator.

1.7 Ranges

Exercise 1.14. Write a range-based code that tests

$$\forall_{\text{prime } p} : \exists_{\text{prime } q} : q > p$$

Exercise 1.15. Rewrite exercise 1.10, using only range expressions, and no loops.

Exercise 1.16. In the above Goldbach exercises you probably needed two prime number sequences, that, however, did not start at the same number. Can you make it so that your code reads

```
all_of( primes_from(5) /* et cetera */
```

1.8 Other

The following exercise requires `std::optional`, which you can learn about in section 24.6.2 (textbook).

Exercise 1.17. Write a function `first_factor` that optionally returns the smallest factor of a given input.

```
auto factor = first_factor(number);
if (factor.has_value())
  cout << "Found factor: " << factor.value() << '\n';
```

1.9 Eratosthenes sieve

The Eratosthenes sieve is an algorithm for prime numbers that step-by-step filters out the multiples of any prime it finds.

1. Start with the integers from 2: $2, 3, 4, 5, 6, \ldots$
2. The first number, 2, is a prime: record it and remove all multiples, giving

   $$3, 5, 7, 9.11, 13, 15, 17 \ldots$$

3. The first remaining number, 3, is a prime: record it and remove all multiples, giving

   $$5, 7, 11, 13, 17, 19, 23, 25, 29 \ldots$$

4. The first remaining number, 5, is a prime: record it and remove all multiples, giving

   $$7, 11, 13, 17, 19, 23, 29, \ldots$$

1.9.1 Arrays implementation

The sieve can be implemented with an array that stores all integers.

Exercise 1.18. Read in an integer that denotes the largest number you want to test. Make an array of integers that long. Set the elements to the successive integers. Apply the sieve algorithm to find the prime numbers.

1.9.2 Streams implementation

The disadvantage of using an array is that we need to allocate an array. What's more, the size is determined by how many integers we are going to test, not how many prime numbers we want to generate. We are going to take the idea above of having a generator object, and apply that to the sieve algorithm: we will now have multiple generator objects, each taking the previous as input and erasing certain multiples from it.

Exercise 1.19. Write a `stream` class that generates integers and use it through a pointer.

Code:

```
for (int i=0; i<7; i++)
  cout << "Next int: "
       << the_ints->next() << '\n';
```

Output
[sieve] ints:

```
Next int: 2
Next int: 3
Next int: 4
Next int: 5
Next int: 6
Next int: 7
Next int: 8
```

Next, we need a stream that takes another stream as input, and filters out values from it.

Exercise 1.20. Write a class `filtered_stream` with a constructor

```
filtered_stream(int filter,shared_ptr<stream> input);
```

that

1. Implements `next`, giving filtered values,
2. by calling the `next` method of the input stream and filtering out values.

Code:

```
auto integers =
  make_shared<stream>();
auto odds =
  shared_ptr<stream>
  ( new filtered_stream(2,integers) );
for (int step=0; step<5; step++)
  cout << "next odd: "
       << odds->next() << '\n';
```

Output
[sieve] odds:

```
next odd: 3
next odd: 5
next odd: 7
next odd: 9
next odd: 11
```

Now you can implement the Eratosthenes sieve by making a `filtered_stream` for each prime number.

Exercise 1.21. Write a program that generates prime numbers as follows.

- Maintain a `current` stream, that is initially the stream of prime numbers.
- Repeatedly:
  - Record the first item from the current stream, which is a new prime number;
  - and set `current` to a new stream that takes `current` as input, filtering out multiples of the prime number just found.

1.10 Range implementation

If we write the prime number definition

$$\begin{aligned} D(n,d) &\equiv n|d = 0 \\ P(n) &\equiv \forall_{d\leq\sqrt{N}}\colon \neg D(n,d) \end{aligned}$$

we see that this involves two streams that we iterate over:

1. First there is the set of all d such that $d^2 \leq n$; then

2. We have the set of booleans testing whether these d values are divisors.

Exercise 1.22. Use the *iota* range view to generate all integers from 2 to infinity, and find a range view that cuts off the seqwuence at the last possible divisor.

Then use the *all_of* or *any_of* rangified algorithms to test whether any of these potential divisors are actually a divisor, and therefore whether or not your number is prime.

Exercise 1.23. Use the *filter* view to filter from an *iota* view those elements that are prime.

Exercise 1.24. Make a *primes* class that can be ranged:

Code:

```
primegenerator allprimes;
for ( auto p : allprimes ) {
  cout << p << ", ";
  if (p>100) break;
}
cout << '\n';
```

Output

```
[primes] range:

missing snippet
../code/primes/range.runout
```

1.11 User-friendliness

Use the *cxxopts* package (section 63.2 (textbook)) to add commandline options to some primality programs.

Exercise 1.25. Take your old prime number testing program, and add commandline options:

- the `-h` option should print out usage information;
- specifying a single int `--test 1001` should print out all primes under that number;
- specifying a set of ints `--tests 57,125,1001` should test primeness for those.

Chapter 2

Geometry

In this set of exercises you will write a small 'geometry' package: code that manipulates points, lines, shapes. These exercises mostly use the material of section 9 (textbook).

2.1 Basic functions

Exercise 2.1. Write a function with (float or double) inputs x, y that returns the distance of point (x, y) to the origin.

Test the following pairs: $1, 0$; $0, 1$; $1, 1$; $3, 4$.

Exercise 2.2. Write a function with inputs x, y, θ that alters x and y corresponding to rotating the point (x, y) over an angle θ.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Your code should behave like:

Code:

```
1 const float pi = 2*acos(0.0);
2 float x{1.}, y{0.};
3 rotate(x,y,pi/4);
4 cout << "Rotated halfway: ("
5      << x << "," << y << ")" << '\n';
6 rotate(x,y,pi/4);
7 cout << "Rotated to the y-axis: ("
8      << x << "," << y << ")" << '\n';
```

Output

```
[geom] rotate:

Rotated halfway:
    (0.707107,0.707107)
Rotated to the y-axis: (0,1)
```

2.2 Point class

A class can contain elementary data. In this section you will make a `Point` class that models Cartesian coordinates and functions defined on coordinates.

Exercise 2.3. Make class `Point` with a constructor

```
Point( float xcoordinate, float ycoordinate );
```

Write the following methods:

- `distance_to_origin` returns a `float`.
- `angle` computes the angle of vector (x, y) with the x-axis.

Exercise 2.4. Extend the `Point` class of the previous exercise with a method: `distance` that computes the distance between this point and another: if `p,q` are `Point` objects,

```
p.distance(q)
```

computes the distance between them.

Exercise 2.5. Write a method *halfway* that, given two `Point` objects `p,q`, construct the `Point` halfway, that is, $(p + q)/2$:

```
Point p(1,2.2), q(3.4,5.6);
Point h = p.halfway(q);
```

You can write this function directly, or you could write functions `Add` and `Scale` and combine these. (Later you will learn about operator overloading.)

How would you print out a *Point* to make sure you compute the halfway point correctly?

Exercise 2.6. Make a default constructor for the point class:

```
Point() { /* default code */ }
```

which you can use as:

```
Point p;
```

but which gives an indication that it is undefined:

Code:

```
Point p3;
cout << "Uninitialized point:"
     << '\n';
p3.printout();
cout << "Using uninitialized point:"
     << '\n';
auto p4 = Point(4,5)+p3;
p4.printout();
```

Output
[geom] linearnan:

```
Uninitialized point:
Point: nan,nan
Using uninitialized point:
Point: nan,nan
```

Hint: see section 26.3.3 (textbook).

Exercise 2.7. Revisit exercise 46.2 (textbook) using the `Point` class. Your code should now look like:

```
newpoint = point.rotate(alpha);
```

Exercise 2.8. Advanced. Can you make a `Point` class that can accommodate any number of space dimensions? Hint: use a `vector`; section 10.3 (textbook). Can you make a constructor where you do not specify the space dimension explicitly?

2.3 Using one class in another

Exercise 2.9. Make a class `LinearFunction` with a constructor:

```
LinearFunction( Point input_p1,Point input_p2 );
```

and a member function

```
float evaluate_at( float x );
```

which you can use as:

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

Exercise 2.10. Make a class `LinearFunction` with two constructors:

```
LinearFunction( Point input_p2 );
LinearFunction( Point input_p1,Point input_p2 );
```

where the first stands for a line through the origin.
Implement again the `evaluate` function so that

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

Exercise 2.11. Revisit exercises 46.2 (textbook) and 46.7 (textbook), introducing a `Matrix` class. Your code can now look like

```
newpoint = point.apply(rotation_matrix);
```

or

```
newpoint = rotation_matrix.apply(point);
```

Can you argue in favor of either one?

Suppose you want to write a `Rectangle` class, which could have methods such as `float Rectangle::area()` or `bool Rectangle::contains(Point)`. Since rectangle has four corners, you could store four `Point` objects in each `Rectangle` object. However, there is redundancy there: you only need three points to infer the fourth. Let's consider the case of a rectangle with sides that are horizontal and vertical; then you need only two points.

Intended API:

```
float Rectangle::area();
```

It would be convenient to store width and height; for

```
bool Rectangle::contains(Point);
```

it would be convenient to store bottomleft/topright points.

Exercise 2.12.

1. Make a class `Rectangle` (sides parallel to axes) with a constructor:

   ```
   Rectangle(Point botleft,float width,float height);
   ```

   The logical implementation is to store these quantities. Implement methods:

   ```
   float area(); float rightedge_x(); float topedge_y();
   ```

   and write a main program to test these.
2. Add a second constructor

   ```
   Rectangle(Point botleft,Point topright);
   ```

   Can you figure out how to use *member initializer* lists for the constructors?

Exercise 2.13. Make a copy of your solution of the previous exercise, and redesign your class so that it stores two `Point` objects. Your main program should not change.

The previous exercise illustrates an important point: for well designed classes you can change the implementation (for instance motivated by efficiency) while the program that uses the class does not change.

2.4 Is-a relationship

Exercise 2.14. Take your code where a `Rectangle` was defined from one point, width, and height.

Make a class `Square` that inherits from `Rectangle`. It should have the function `area` defined, inherited from `Rectangle`.

First ask yourself: what should the constructor of a `Square` look like?

Exercise 2.15. Revisit the `LinearFunction` class. Add methods `slope` and `intercept`.

Now generalize `LinearFunction` to `StraightLine` class. These two are almost the same except for vertical lines. The `slope` and `intercept` do not apply to vertical lines, so design `StraightLine` so that it stores the defining points internally. Let `LinearFunction` inherit.

2.5 Pointers

The following exercise is a little artificial.

Exercise 2.16. Make a *DynRectangle* class, which is constructed from two shared-pointers-to-*Point* objects:

```
auto
  origin  = make_shared<Point>(0,0),
  fivetwo = make_shared<Point>(5,2);
DynRectangle lielow( origin,fivetwo );
```

Calculate the area, scale the top-right point, and recalculate the area:

Code:

```
1 cout << "Area: " <<  lielow.area() <<
      '\n';
2  /* ... */
3 // scale the 'fivetwo' point by two
4 cout << "Area: " <<  lielow.area() <<
      '\n';
```

Output
[pointer] dynrect:

```
Area: 10
Area: 40
```

You can base this off the file `pointrectangle.cpp` *in the repository*

2.6 More stuff

The `Rectangle` class stores at most one corner, but may be convenient to sometimes have an array of all four corners.

Exercise 2.17. Add a method

```
const vector<Point> &corners()
```

to the `Rectangle` class. The result is an array of all four corners, not in any order. Show by a compiler error that the array can not be altered.

Exercise 2.18. Revisit exercise 2.5 and replace the *add* and *scale* functions by overloaded operators.

Hint: for the *add* function you may need '**this**'.

Chapter 3

Zero finding

3.1 Root finding by bisection

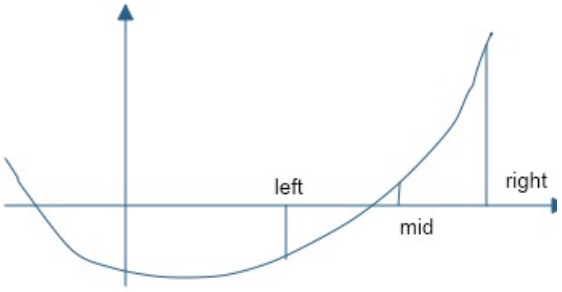


Figure 3.1: Root finding by interval bisection

For many functions f, finding their zeros, that is, the values x for which $f(x) = 0$, can not be done analytically. You then have to resort to numerical *root finding* schemes. In this project you will develop gradually more complicated implementations of a simple scheme: root finding by *bisection*.

In this scheme, you start with two points where the function has opposite signs, and move either the left or right point to the mid point, depending on what sign the function has there. See figure 3.1.

In section 3.2 we will then look at Newton's method.

Here we will not be interested in mathematical differences between the methods, though these are important: we will use these methods to exercise some programming techniques.

3.1.1 Simple implementation

Let's develop a first implementation step by step. To ensure correctness of our code we will use a Test-Driven Development (TDD) approach: for each bit of functionality we write a test to ensure its correctness before we integrate it in the larger code. (For more about TDD, and in particular the Catch2 framework, see section 68.2 (textbook).)

3.1.2 Polynomials

First of all, we need to have a way to represent polynomials. For a polynomial of degree d we need $d+1$ coefficients:

$$f(x) = c_0 x^d + \cdots + c_{d-1} x^1 + c_d \tag{3.1}$$

We implement this by storing the coefficients in a `vector<double>`. We make the following arbitrary decisions

1. let the first element of this vector be the coefficient of the highest power, and
2. for the coefficients to properly define a polynomial, this leading coefficient has to be nonzero.

Let's start by having a fixed test polynomial, provided by a function `set_coefficients`. For this function to provide a proper polynomial, it has to satisfy the following test:

```
TEST_CASE( "coefficients represent polynomial" "[1]") {
  vector<double> coefficients = { 1.5, 0., -3 };
  REQUIRE( coefficients.size()>0 );
  REQUIRE( coefficients.front()!=0. );
}
```

Exercise 3.1. Write a routine `set_coefficients` that constructs a vector of coefficients:

```
vector<double> coefficients = set_coefficients();
```

and make it satisfy the above conditions.

At first write a hard-coded set of coefficients, then try reading them from the command line.

Above we postulated two conditions that an array of numbers should satisfy to qualify as the coefficients of a polynomial. Your code will probably be testing for this, so let's introduce a boolean function `is_proper_polynomial`:

- This function returns `true` if the array of numbers satisfies the two conditions;
- it returns `false` if either condition is not satisfied.

In order to test your function `is_proper_polynomial` you should check that

- it recognizes correct polynomials, and
- it fails for improper coefficients that do not properly define a polynomial.

Exercise 3.2. Write a function `is_proper_polynomial` as described, and write unit tests for it, both passing and failing:

```
vector<double> good = /* proper coefficients */ ;
REQUIRE( is_proper_polynomial(good) );
vector<double> notso = /* improper coefficients */ ;
REQUIRE( not is_proper_polynomial(good) );
```

Next we need polynomial evaluation. We will build a function `evaluate_at` with the following definition:

```
double evaluate_at( const std::vector<double>& coefficients,double x);
```

You can interpret the array of coefficients in (at least) two ways, but with equation (3.1) we proscribed one particular interpretation.

So we need a test that the coefficients are indeed interpreted with the leading coefficient first, and not with the leading coefficient last. For instance:

```
polynomial second( {2,0,1} );
// correct interpretation: 2x^2 + 1
REQUIRE( second.is_proper() );
REQUIRE( second.evaluate_at(2) == Catch::Approx(9) );
// wrong interpretation: 1x^2 + 2
REQUIRE( second.evaluate_at(2) != Catch::Approx(6) );
```

(where we have left out the `TEST_CASE` header.)

Now we write the function that passes these tests:

Exercise 3.3. Write a function `evaluate_at` which computes

$$y \leftarrow f(x).$$

and confirm that it passes the above tests.

```
double evaluate_at( polynomial coefficients,double x);
```

For bonus points, look up *Horner's rule* and implement it.

With the polynomial function implemented, we can start working towards the algorithm.

3.1.3 Left/right search points

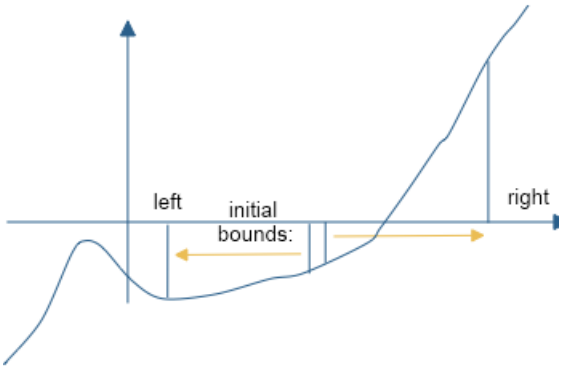


Figure 3.2: Setting the initial search points

Suppose x_-, x_+ are such that

$$x_- < x_+, \quad \text{and} \quad f(x_-) \cdot f(x_+) < 0,$$

that is, the function values in the left and right point are of opposite sign. Then there is a zero in the interval (x_-, x_+); see figure 3.1.

But how to find these outer bounds on the search?

If the polynomial is of odd degree you can find x_-, x_+ by going far enough to the left and right from any two starting points. For even degree there is no such simple algorithm (indeed, there may not be a zero) so we abandon the attempt.

We start by writing a function `is_odd` that tests whether the polynomial is of odd degree.

Exercise 3.4. Make the following code work:

```
if ( not is_odd(coefficients) ) {
  cout << "This program only works for odd-degree polynomials\n";
  exit(1);
}
```

You could test the above as:

```
polynomial second{2,0,1}; // 2x^2 + 1
REQUIRE( not is_odd(second) );
polynomial third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE( is_odd(third) );
```

Now we can find x_-, x_+: start with some interval and move the end points out until the function values have opposite sign.

Exercise 3.5. Write a function `find_initial_bounds` which computes x_-, x_+ such that

$$f(x_-) < 0 < f(x_+) \quad \text{or} \quad f(x_+) < 0 < f(x_-)$$

How can you compute this test more compactly?
What is a good prototype for the function?
How do move the points far enough out to satisfy this condition?

Since finding a left and right point with a zero in between is not always possible for polynomials of even degree, we completely reject this case. In the following test we throw an exception (see section 23.2.2 (textbook), in particularly 23.2.2.3 (textbook)) for polynomials of even degree:

```
right = left+1;
polynomial second{2,0,1}; // 2x^2 + 1
REQUIRE_THROWS( find_initial_bounds(second,left,right) );
polynomial third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE_NOTHROW( find_initial_bounds(third,left,right) );
REQUIRE( left<right );
```

Make sure your code passes these tests. What test do you need to add for the function values?

3.1.4 Root finding

The root finding process globally looks as follows:

- You start with points x_-, x_+ where the function has opposite sign; then you know that there is a zero between them.
- The bisection method for finding this zero looks at the halfway point, and based on the function value in the mid point:

- moves one of the bounds to the mid point, such that the function again has opposite signs in the left and right search point.

The structure of the code is as follows:

```
double find_zero( /* something */ ) {
  while ( /* left and right too far apart */ ) {
    // move bounds left and right closer together
  }
  return something;
}
```

Again, we test all the functionality separately. In this case this means that moving the bounds should be a testable step.

Exercise 3.6. Write a function `move_bounds_closer` and test it.

```
void move_bounds_closer
   ( std::vector<double> coefficients,
     double& left,double& right );
```

Implement some unit tests on this function.

Finally, we put everything together in the top level function `find_zero`.

Exercise 3.7. Make this call work:

```
auto zero = find_zero( coefficients, 1.e-8 );
cout << "Found root " << zero
     << " with value " << evaluate_at(coefficients,zero) << '\n';
```

Design unit tests, including on the precision attained, and make sure your code passes them.

3.1.5 Object implementation

Revisit the exercises of section 3.1.1 and introduce a `polynomial` class that stores the polynomial coefficients. Several functions now become members of this class.

Also update the unit tests.

How can you generalize the polynomial class, for instance to the case of special forms such as $(1+x)^n$?

3.1.6 Templating

In the implementations so far we used `double` for the numerical type. Make a templated version that works both with `float` and `double`.

Can you see a difference in attainable precision between the two types?

3.2 Newton's method

In this section we look at Newton's method. This is an iterative method for finding zeros of a function f, that is, it computes a sequence of values $\{x_n\}_n$, so that $f(x_n) \to 0$. The sequence is defined by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

with x_0 arbitrarily chosen.

Early computers had no hardware for computing a square root. Instead, they used *Newton's method*; for details, see HPC book [6], section 17.

Suppose you have a value y and you want want to compute $x = \sqrt{y}$. This is equivalent to finding the zero of

$$f(x) = x^2 - y$$

where y is fixed. To indicate this dependence on y, we will write $f_y(x)$. Newton's method then finds the zero by evaluating

$$x_{\text{next}} = x - f_y(x)/f_y'(x)$$

until the guess is accurate enough, that is, until $f_y(x) \approx 0$.

3.2.1 Function implementation

It is of course simple to code this specific case; it should take you about 10 lines. However, we want to have a general code that takes any two functions f, f', and then uses Newton's method to find a zero of f.

Exercise 3.8.

- Write functions `f(x,y)` and `deriv(x,y)`, that compute $f_y(x)$ and $f_y'(x)$ for the definition of f_y above.
- Read a value y and iterate until $|\texttt{f}(\texttt{x},\texttt{y})| < 10^{-5}$. Print x.
- Second part: write a function `newton_root` that computes $\sqrt{y}$.

3.2.2 Using lambdas

Above you wrote functions conforming to:

```
double f(double x) { return x*x-2; };
double fprime(double x) { return 2*x; };
```

and the algorithm:

```
double x{1.};
while ( true ) {
  auto fx = f(x);
  cout << "f( " << x << " ) = " << fx << '\n';
  if (std::abs(fx)<1.e-10 ) break;
  x = x - fx/fprime(x);
}
```

Exercise 3.9. Rewrite your code to use lambda functions for `f` and `fprime`.

You can base this off the file `newton.cpp` in the repository

Next, we make the code modular by writing a general function `newton_root`, that contains the Newton method of the previous exercise. Since it has to work for any functions f, f', you have to pass the objective function and the derivative as arguments:

```
double root = newton_root( f,fprime );
```

Exercise 3.10. Rewrite the Newton exercise above to use a function that is used as:

```
double root = newton_root( f,fprime );
```

Call the function

1. first with the lambda variables you already created;
2. but in a better variant, directly with the lambda expressions as arguments, that is, without assigning them to variables.

Next we extend functionality, but not by changing the root finding function: instead, we use a more general way of specifying the objective function and derivative.

Exercise 3.11. Extend the newton exercise to compute roots in a loop:

```
for (int n=2; n<=8; n++) {
  cout << "sqrt(" << n << ") = "
       << newton_root(
  /* ... */
                      )
       << '\n';
```

Without lambdas, you would define a function

```
double squared_minus_n( double x,int n ) {
  return x*x-n; }
```

However, the `newton_root` function takes a function of only a real argument. Use a capture to make `f` dependent on the integer parameter.

Exercise 3.12. You don't need the gradient as an explicit function: you can approximate it as

$$f'(x) = \big(f(x+h) - f(x)\big)/h$$

for some value of h.

Write a version of the root finding function

```
double newton_root( function< double(double)> f )
```

that uses this. You can use a fixed value `h=1e-6`. Do not reimplement the whole newton method: instead create a lambda for the gradient and pass it to the function `newton_root` you coded earlier.

Exercise 3.13. Bonus: can you compute logarithms through Newton's method?

3.2.3 Templated implementation

Newton's method works equally well for complex numbers as for real numbers.

Exercise 3.14. Rewrite your Newton program so that it works for complex numbers:

```
complex<double> z{.5,.5};
while ( true ) {
  auto fz = f(z);
  cout << "f( " << z << " ) = " << fz << '\n';
  if (std::abs(fz)<1.e-10 ) break;
  z = z - fz/fprime(z);
}
```

You may run into the problem that you can not operate immediately between a complex number and an integer. Use `static_cast`.

So do you have to write two separate implementations, one for reals, and one for complex numbers? (And maybe you need two separate ones for `float` and `double`!)

This is where *templates* come in handy; chapter 22 (textbook).

You can templatize your Newton function and derivative:

```
template<typename T>
T f(T x) { return x*x - 2; };
template<typename T>
T fprime(T x) { return 2 * x; };
```

and then write

```
double x{1.};
while ( true ) {
  auto fx = f<double>(x);
  cout << "f( " << x << " ) = " << fx << '\n';
  if (std::abs(fx)<1.e-10 ) break;
  x = x - fx/fprime<double>(x);
}
```

Exercise 3.15. Update your Newton program with templates. If you have it working for `double`, try using `complex<double>`. Does it work?

Exercise 3.16. Use your complex Newton method to compute $\sqrt{2}$. Does it work?

How about $\sqrt{-2}$?

Exercise 3.17. Can you templatize your Newton code that used lambda expressions? Your function header would now be:

```
template<typename T>
```

```
T newton_root
   ( function< T(T) > f,
     function< T(T) > fprime,
     T init) {
```

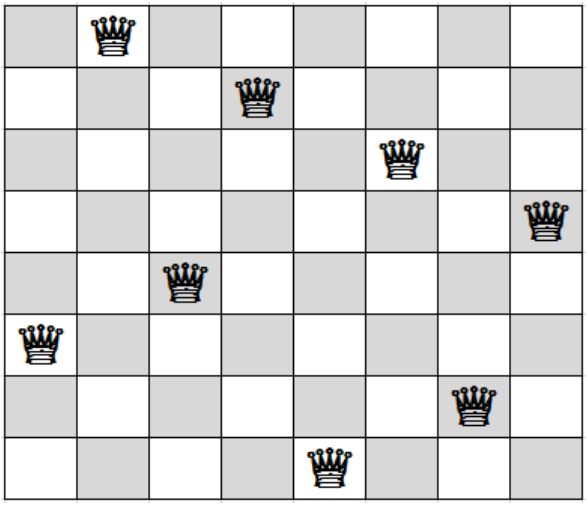
You would for instance compute $\sqrt{2}$ as:

```
cout << "sqrt -2 = " <<
     newton_root<complex<double>>
          ( [] (complex<double> x) {
               return x*x + static_cast<complex<double>>(2); },
            [] (complex<double> x) {
              return x * static_cast<complex<double>>(2); },
            complex<double>{.1,.1}
            )
     << '\n';
```

Chapter 4

Eight queens

A famous exercise for recursive programming is the *eight queens* problem: Is it possible to position eight queens on a chess board, so that no two queens 'threaten' each other, according to the rules of chess?



4.1 Problem statement

The precise statement of the 'eight queens problem' is:

- Put eight pieces on an 8×8 board, no two pieces on the same square; so that
- no two pieces are on the same row,
- no two pieces are on the same column, and
- no two pieces are on the same diagonal.

A systematic solution would run:

1. put a piece anywhere in the first row;
2. for each choice in the first row, try all positions in the second row;
3. for all choices in the first two rows, try all positions in the third row;
4. when you have a piece in all eight rows, evaluate the board to see if it satisfies the condition.

Exercise 4.1. This algorithm will generate all 8^8 boards. Do you see at least one way to speed up the search?

Since the number eight is, for now, fixed, you could write this code as an eight-deep loop nest. However that is not elegant. For example, the only reason for the number 8 in the above exposition is that this is the traditional size of a chess board. The problem, stated more abstractly as placing n queens on an $n \times n$ board, has solutions for $n \geq 4$.

4.2 Solving the eight queens problem, basic approach

This problem requires you to know about arrays/vectors; chapter 10 (textbook). Also, see chapter 68 (textbook) for TDD. Finally, see section 24.6.2 (textbook) for `std::optional`.

The basic strategy will be that we fill consecutive rows, by indicating each time which column will be occupied by the next queen. Using an Object-Oriented (OO) strategy, we make a class `ChessBoard`, that holds a partially filled board.

The basic solution strategy is recursive:

- Let `current` be the current, partially filled board;
- We call `current.place_queens()` that tries to finish the board;
- However, with recursion, this method only fills one row, and then calls `place_queens` on this new board.

So

```
ChessBoard::place_queens() {
  // for c = 1 ... number of columns:
  //   make a copy of the board
  //   put a queen in the next row, column c, of the copy
  //   and call place_queens() on that copy;
  //   investigate the result.....
}
```

This routine returns either a solution, or an indication that no solution was possible.

In the next section we will develop a solution systematically in a TDD manner.

4.3 Developing a solution by TDD

We now gradually develop the OO solution to the eight queens problem, using test-driven development.

The board We start by constructing a board, with a constructor that only indicates the size of the problem:

```
ChessBoard(int n);
```

This is a 'generalized chess board' of size $n \times n$, and initially it should be empty.

Exercise 4.2. Write this constructor, for an empty board of size $n \times n$.

Note that the implementation of the board is totally up to you. In the following you will get tests for functionality that you need to satisfy, but any implementation that makes this true is a correct solution.

Bookkeeping: what's the next row? Assuming that we fill in the board row-by-row, we have an auxiliary function that returns the next row to be filled:

```
int next_row_to_be_filled()
```

This gives us our first simple test: on an empty board, the row to be filled in is row zero.

Exercise 4.3. Write this method and make sure that it passes the test for an empty board.

```
TEST_CASE( "empty board","[1]" ) {
  constexpr int n=10;
  ChessBoard empty(n);
  REQUIRE( empty.next_row_to_be_filled()==0 );
}
```

By the rules of TDD you can actually write the method so that it only satisfies the test for the empty board. Later, we will test that this method gives the right result after we have filled in a couple of rows, and then of course your implementation needs to be general.

Place one queen Next, we have a function to place the next queen, whether this gives a feasible board (meaning that no pieces can capture each other) or not:

```
void place_next_queen_at_column(int i);
```

This method should first of all catch incorrect indexing: we assume that the placement routine throws an exception for invalid column numbers.

```
ChessBoard::place_next_queen_at_column( int c ) {
  if ( /* c is outside the board */ )
    throw(1); // or some other exception.
```

(Suppose you didn't test for incorrect indexing. Can you construct a simple 'cheating' solution at any size?)

Exercise 4.4. Write this method, and make sure it passes the following test for valid and invalid column numbers:

```
REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
REQUIRE( empty.next_row_to_be_filled()==1 );
```

(From now on we'll only give the body of the test.)

Now it's time to start writing some serious stuff.

Is a (partial) board feasible? If you have a board, even partial, you want to test if it's feasible, meaning that the queens that have been placed can not capture each other.

The prototype of this method is:

```
bool feasible()
```

This test has to work for simple cases to begin with: an empty board is feasible, as is a board with only one piece.

```
ChessBoard empty(n);
REQUIRE( empty.feasible() );
```

```
ChessBoard one = empty;
one.place_next_queen_at_column(0);
REQUIRE( one.next_row_to_be_filled()==1 );
REQUIRE( one.feasible() );
```

Exercise 4.5. Write the method and make sure it passes these tests.

We shouldn't only do successful tests, sometimes referred to as the 'happy path' through the code. For instance, if we put two queens in the same column, the test should fail.

Exercise 4.6. Take the above initial attempt with a queen in position $(0, 0)$, and add another queen in column zero of the next row. Check that it passes the test:

```
ChessBoard collide = one;
// place a queen in a 'colliding' location
collide.place_next_queen_at_column(0);
// and test that this is not feasible
REQUIRE( not collide.feasible() );
```

Add a few tests more of your own. (These will not be exercised by the submission script, but you may find them useful anyway.)

Testing configurations If we want to test the feasibility of non-trivial configurations, it is a good idea to be able to 'create' solutions. For this we need a second type of constructor where we construct a fully filled chess board from the locations of the pieces.

```
ChessBoard( int n,vector<int> cols );
ChessBoard( vector<int> cols );
```

- If the constructor is called with only a vector, this describes a full board.
- Adding an integer parameter indicates the size of the board, and the vector describes only the rows that have been filled in.

Exercise 4.7. Write these constructors, and test that an explicitly given solution is a feasible board:

```
ChessBoard five( {0,3,1,4,2} );
REQUIRE( five.feasible() );
```

For an elegant approach to implementing this, see *delegating constructors*; section 9.4.1 (textbook).

Ultimately we have to write the tricky stuff.

4.4 The recursive solution method

The main function

```
optional<ChessBoard> place_queens()
```

takes a board, empty or not, and tries to fill the remaining rows.

One problem is that this method needs to be able to communicate that, given some initial configuration, no solution is possible. For this, we let the return type of `place_queens` be `optional<ChessBoard>`:

- if it is possible to finish the current board resulting in a solution, we return that filled board;
- otherwise we return `{}`, indicating that no solution was possible.

With the recursive strategy discussed in section 4.2, this placement method has roughly the following structure:

```
place_queens() {
  for ( int col=0; col<n; col++ ) {
    ChessBoard next = *this;
    // put a queen in column col on the 'next' board
    // if this is feasible and full, we have a solution
    // if it is feasible but no full, recurse
  }
}
```

The line

```
ChessBoard next = *this;
```

makes a copy of the object you're in.

Remark 1 *Another approach would be to make a recursive function*

```
bool place_queen( const ChessBoard& current, ChessBoard &next );
// true if possible, false is not
```

The final step Above you coded the method `feasible` that tested whether a board is still a candidate for a solution. Since this routine works for any partially filled board, you also need a method to test if you're done.

Exercise 4.8. Write a method

```
bool filled();
```

and write a test for it, both positive and negative.

Now that you can recognize solutions, it's time to write the solution routine.

Exercise 4.9. Write the method

```
optional<ChessBoard> place_queens()
```

Because the function `place_queens` is recursive, it is a little hard to test in its entirety.

We start with a simpler test: if you almost have the solution, it can do the last step.

Exercise 4.10. Use the constructor

```
ChessBoard( int n,vector<int> cols )
```

to generate a board that has all but the last row filled in, and that is still feasible. Test that you can find the solution:

```
ChessBoard almost( 4, {1,3,0} );
```

```
auto solution = almost.place_queens();
REQUIRE( solution.has_value() );
REQUIRE( solution->filled() );
```

Since this test only fills in the last row, it only does one loop, so printing out diagnostics is possible, without getting overwhelmed in tons of output.

Solutions and non-solutions Now that you have the solution routine, test that it works starting from an empty board. For instance, confirm there are no 3×3 solutions:

```
TEST_CASE( "no 3x3 solutions","[9]" ) {
  ChessBoard three(3);
  auto solution = three.place_queens();
  REQUIRE( not solution.has_value() );
}
```

On the other hand, 4×4 solutions do exist:

```
TEST_CASE( "there are 4x4 solutions","[10]" ) {
  ChessBoard four(4);
  auto solution = four.place_queens();
  REQUIRE( solution.has_value() );
}
```

Exercise 4.11. (Optional) Can you modify your code so that it counts all the possible solutions?

Exercise 4.12. (Optional) How does the time to solution behave as function of n?

PART II

RESEARCH PROJECTS

Chapter 5

Infectuous disease simulation

This section contains a sequence of exercises that builds up to a somewhat realistic simulation of the spread of infectious diseases.

5.1 Model design

It is possible to model disease propagation statistically, but here we will build an explicit simulation: we will maintain an explicit description of all the people in the population, and track for each of them their status.

We will use a simple model where a person can be:

- sick: when they are sick, they can infect other people;
- susceptible: they are healthy, but can be infected;
- recovered: they have been sick, but no longer carry the disease, and can not be infected for a second time;
- vaccinated: they are healthy, do not carry the disease, and can not be infected.

In more complicated models a person could be infectious during only part of their illness, or there could be secondary infections with other diseases, et cetera. We keep it simple here: any sick person can infect others while they are sick.

In the exercises below we will gradually develop a model of how the disease spreads from an initial source of infection. The program will then track the population from day to day, running indefinitely until none of the population is sick. Since there is no re-infection, the run will always end. Later we will add mutation to the model which can extend the duration of the epidemic.

5.1.1 Other ways of modeling

Instead of capturing every single person in code, a 'contact network' model, it is possible to use an Ordinary Diffential Equation (ODE) approach to disease modeling. You would then model the percentage of infected persons by a single scalar, and derive relations for that and other scalars [2, 9].

This is known as a 'compartmental model', where each of the three SIR states, Susceptible, Infected, Removed, is a compartment: a section of the population. Both the contact network and the compartmental model capture part of the truth. In fact, they can be combined. We can consider a country as a set of cities,

where people travel between any pair of cities. We then use a compartmental model inside a city, and a contact network between cities.

In this project we will only use the network model.

5.2 Coding

The following sections are a step-by-step development of the code for this project. Later we will discuss running this code as a scientific experiment.

Remark 2 *In various places you need a random number generator. You can use the C language random number generator (section 24.7.5 (textbook)), or the new Standard Template Library (STL) one in section 24.7 (textbook).*

5.2.1 Person basics

The most basic component of a disease simulation is to infect a person with a disease, and see the time development of that infection. Thus you need a person class and a disease class.

Before you start coding, ask yourself what behaviors these classes need to support.

- Person. The basic methods for a person are
  1. Get infected;
  2. Get vaccinated; and
  3. Progress by one day.

  Furthermore you may want to query what the state of the person is: are they healthy, sick, recovered?
- Disease. For now, a disease itself doesn't do much. (Later in the project you may want it to have a method `mutate`.) However, you may want to query certain properties:
  1. Chance of transmission; and
  2. Number of days a person stays sick when infected.

A test for a single person could have output along the following lines:

```
On day 10, Joe is susceptible
On day 11, Joe is susceptible
On day 12, Joe is susceptible
On day 13, Joe is susceptible
On day 14, Joe is sick (5 days to go)
On day 15, Joe is sick (4 days to go)
On day 16, Joe is sick (3 days to go)
On day 17, Joe is sick (2 days to go)
On day 18, Joe is sick (1 days to go)
On day 19, Joe is recovered
```

Exercise 5.1. Write a `Person` class with methods:

- `status_string()` : returns a description of the person's state as a `string`;
- `one_more_day()` : update the person's status to the next day;
- `infect(s)` : infect a person with a disease, where the disease object

```
Disease s(n);
```

is specified to run for n days.

Your main program could for instance look like:

```
for ( int step = 1; ; step++) {

  joe.one_more_day();
  /* ... */
  cout << "On day " << step << ", Joe is "
       << joe.status_string() << '\n';
  if (joe.is_recovered())
    break;
}
```

where the infection part has been left out.

Your main concern is how to model the internal state of a person. This is actually two separate issues:

1. the state, and
2. if sick, how many days to go to recover.

You can find a way to implement this with a single integer, but it's better to use two. Also, write enough support methods such as the `is_recovered` test.

5.2.1.1 *Person tests*

It is easy to write code that seems to be the right thing, but does not behave correctly in all cases. So it is a good idea to subject your code to some systematic tests.

Make sure your `Person` objects pass these tests:

- After being infected with a 100% transmittable disease, they should register as sick.
- If they are vaccinated or recovered, and they come in contact with such a disease, they stay in their original state.
- If a disease has a transmission chance of 50%, and a number of people come into contact with it, about half of them should get sick. This test maybe a little tricky to write.

Can you use the *Catch2* unittesting framework? See section 63.3 (textbook).

5.2.2 Interaction

Next we model interactions between people: one person is healthy, another is infected, and when the two come into contact the disease may be transferred.

```
Person infected,healthy;
infected.infect(flu);
/* ... */
  healthy.touch(infected);
```

The disease has a certain probability of being transferred, so you need to specify that probability. You could let the declaration be:

```
Disease flu( 5, 0.3 );
```

where the first parameter is the number of days an infection lasts, and the second the transfer probability.

Exercise 5.2. Add a transmission probability to the `Disease` class, and add a `touch` method to the `Person` class. Design and run some tests.

Exercise 5.3. Bonus: can you get the following disease specification to work?

```
Disease flu;
flu.duration() = 20;
flu.transfer_probability() = p;
```

Why could you consider this better than the earlier suggested syntax?

5.2.2.1 Interaction tests

Adapt the above tests, but now a person comes in contact with an infected person, rather than directly with a disease.

5.2.3 Population

Next we need a `Population` class, where a population contains a `vector` consisting of `Person` objects. Initially we only infect one person, and there is no transmission of the disease.

The `Population` class should at least have the following methods:

- `random_infection` to start out with an infected segment of the population;
- `random_vaccination` to start out with a number of vaccinated individuals.
- counting functions `count_infected` and `count_vaccinated`.

To run a realistic simulation you also need a `one_more_day` method that takes the population through a day. This is the heart of your code, and we will develop this gradually in the next section.

5.2.3.1 Population tests

Most population testing will be done in the following section. For now, make sure you pass the following tests:

- With a vaccination percentage of 100%, everyone should indeed be vaccinated.

5.3 Epidemic simulation

To simulate the spread of a disease through the population, we need an update method that progresses the population through one day:

- Sick people come into contact with a number of other members of the populace;
- and everyone gets one day older, meaning mostly that sick people get one day closer to recovery.

We develop this in a couple of steps.

5.3.1 No contact

At first assume that people have no contact, so the disease ends with the people it starts with.

The trace output should look something like:

```
Size of
population?
In step   1 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   2 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   3 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   4 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   5 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   6 #sick:    0 :  ? ? ? ? ? ? ? ? ? ? - ? ? ? ? ? ? ? ? ?
Disease ran its course by step 6
```

Remark 3 *Such a display is good for a sanity check on your program behavior. If you include such displays in your writeup, make sure to use a monospace font, and don't use a population size that needs line wrapping. In further testing, you should use large populations, but do not include these displays.*

Exercise 5.4. Program a population without infection.

- Write the `Population` class. The constructor takes the number of people:

  ```
  Population population(npeople);
  ```

- Write a method that infects a number of random people:

  ```
  population.random_infection(fever,initial_infect);
  ```

- Write a method `count_infected` that counts how many people are infected.
- Write an `one_more_day` method that updates all persons in the population.
- Loop the `one_more_day` method until no people are infected: the `Population::one_more_day` method should apply `Person::one_more_day` to all person in the population.

Write a routine that displays the state of the popular, using for instance: `?` for susceptible, + for infected, − for recovered.

5.3.1.1 Tests

Test that for the duration of the disease, the number of infected people stays constant, and that the sum of healthy and infected people stays equal to the population size.

5.3.2 Contagion

This past exercise was too simplistic: the original patient zero was the only one who ever got sick. Now let's incorporate contagion, and investigate the spread of the disease from a single infected person.

We start with a very simple model of infection.

Exercise 5.5. Write a simulation where in each step the direct neighbors of an infected person can now get sick themselves.

Run a number of simulations with population sizes and contagion probabilities. Are there cases where people escape getting sick?

5.3.2.1 Tests

Do some sanity tests:

- If one person is infected with a disease with $p = 1$, the next day there should be 3 people sick. Unless the infected person is the first or last: then there are two.
- If person 0 is infected, and $p = 1$, the simulation should run for a number of days equal to the size of the population.
- How is the previous case if $p = 0.5$?

5.3.3 Vaccination

Exercise 5.6. Incorporate vaccination: read another number representing the percentage of people that has been vaccinated. Choose those members of the population randomly.

Describe the effect of vaccinated people on the spread of the disease. Why is this model unrealistic?

5.3.4 Spreading

To make the simulation more realistic, we let every sick person come into contact with a fixed number of random people every day. This gives us more or less the *SIR model*; https://en.wikipedia.org/wiki/Epidemic_model.

Set the number of people that a person comes into contact with, per day, to 6 or so. (You can also let this be an upper bound for a random value, but that does not essentially change the simulation.) You have already programmed the probability that a person who comes in contact with an infected person gets sick themselves. Again start the simulation with a single infected person.

Exercise 5.7. Code the random interactions. Now run a number of simulations varying

- The percentage of people vaccinated, and
- the chance the disease is transmitted on contact.

Record how long the disease runs through the population. With a fixed number of contacts and probability of transmission, how is this number of function of the percentage that is vaccinated?

Report this function as a table or graph. Make sure you have enough data points for a meaningful conclusion. Use a realistic population size. You can also do multiple runs and report the average, to even out the effect of the random number generator.

Exercise 5.8. Investigate the matter of 'herd immunity': if enough people are vaccinated, then some people who are not vaccinated will still never get sick. Let's say you want to have the probability of being not vaccinated, yet never getting sick, to be over 95 percent. Investigate the percentage of vaccination that is needed for this as a function of the contagiousness of the disease.

As in the previous exercise, make sure your data set is large enough.

Remark 4 *The screen output you used above is good for sanity checks on small problems. However, for realistic simulations you have to think what is a realistic population size. If your university campus is a population where random people are likely to meet each other, what would be a population size to model that? How about the city where you live?*

Likewise, if you test different vaccination rates, what granularity do you use? With increases of 5 or 10 percent you can print all results to you screen, but you may miss things. Don't be afraid to generate large amount of data and feed them directly to a graphing program.

5.3.5 Mutation

The Covid years have shown how important mutations of an original virus can be. Next, you can include mutation in your project. We model this as follows:

- Every so many transmissions, a virus will mutate into a new variant.
- A person who has recovered from one variant is still susceptible to other variants.
- For simplicity assume that each variant leaves a person sick the same number of days, and
- Vaccination is all-or-nothing: one vaccine is enough to protect against all variant;
- On the other hand, having recovered from one variant is not protection against others.

Implementation-wise speaking, we model this as follows. First of all, we need a `Disease` class, so that we can infect a person with an explicit virus;

```
void touch( const Person&,long int ps=0 );
void infect( const Disease& );
```

A `Disease` object now carries the information such as the chance of transmission, or how a long a person stays under the weather. Modeling mutation is a little tricky. You could do it as follows:

- There is a global `variants` counter for new virus variants, and a global `transmissions` counter.
- Every time a person infects another, the newly infected person gets a new `Disease` object, with the current variant, and the transmissions counter is updated.
- There is a parameter that determines after how many transmissions the disease mutates. If there is a mutation, the global `variants` counter is updated, and from that point on, every infection is with the new variant. (Note: this is not very realistic. You are free to come up with a better model.)
- A each `Person` object has a vector of variants that they are recovered from; recovery from one variant only makes them immune from that specific variant, not from others.

Exercise 5.9. Add mutation to your model. Experiment with the mutation rate: as the mutation rate increases, the disease should stay in the population longer. Does the relation with vaccination rate change that you observed before?

5.3.6 Diseases without vaccine: Ebola and Covid-19

This section is optional, for bonus points

The project so far applies to diseases for which a vaccine is available, such as MMR for measles, mumps and rubella. The analysis becomes different if no vaccine exists, such as is the case for *Ebola* and *Covid-19*, as of this writing.

Instead, you need to incorporate 'social distancing' into your code: people do not get in touch with random others anymore, but only those in a very limited social circle. Design a model distance function, and explore various settings.

The difference between Ebola and Covid-19 is how long an infection can go unnoticed: the *incubation period*. With Ebola, infection is almost immediately apparent, so such people are removed from the general population and treated in a hospital. For Covid-19, a person can be infected, and infect others, for a number of days before they are sequestered from the population.

Add this parameter to your simulation and explore the behavior of the disease as a function of it.

5.4 Ethics

The subject of infectious diseases and vaccination is full of ethical questions. The main one is *The chances of something happening to me are very small, so why shouldn't I bend the rules a little?*. This reasoning is most often applied to vaccination, where people for some reason or other refuse to get vaccinated.

Explore this question and others you may come up with: it is clear that everyone bending the rules will have disastrous consequences, but what if only a few people did this?

5.5 Project writeup and submission

5.5.1 Program files

In the course of this project you have written more than one main program, but some code is shared between the multiple programs. Organize your code with one file for each main program, and a single 'library' file with the class methods.

You can do this two ways:

1. You make a 'library' file, say `infect_lib.cc`, and your main programs each have a line

   ```
   #include "infect_lib.cc"
   ```

   This is not the best solution, but it is acceptable for now.
2. The better solution requires you to use *separate compilation* for building the program, and you need a *header* file. You would now have `infect_lib.cc` which is compiled separately, and `infect_lib.h` which is included both in the library file and the main program:

   ```
   #include "infect_lib.h"
   ```

   See section 19.2.2 (textbook) for more information.

Submit all source files with instructions on how to build all the main programs. You can put these instructions in a file with a descriptive name such as `README` or `INSTALL`, or you can use a *makefile*.

5.5.2 Writeup

In the writeup, describe the 'experiments' you have performed and the conclusions you draw from them. The exercises above give you a number of questions to address.

For each main program, include some sample output, but note that this is no substitute for writing out your conclusions in full sentences.

The exercises in section 49.3.4 (textbook) ask you to explore the program behavior as a function of one or more parameters. Include a table to report on the behavior you found. You can use Matlab or Matplotlib in Python (or even Excell) to plot your data, but that is not required.

5.6 Bonus: mathematical analysis

The SIR model can also be modeled through coupled difference or differential equations.

1. The number S_i of susceptible people at time i decreases by a fraction

   $$S_{i+1} = S_i(1 - \lambda_i\, dt)$$

   where λ_i is the product of the number of infected people and a constant that reflects the number of meetings and the infectiousness of the disease. We write:

   $$S_{i+1} = S_i(1 - \lambda I_i\, dt)$$

2. The number of infected people similarly increases by $\lambda S_i I_i$, but it also decreases by people recovering (or dying):

   $$I_{i+1} = I_i(1 + \lambda S_i\, dt - \gamma\, dt).$$

3. Finally, the number of 'removed' people equals that last term:

   $$R_{i+1} = R_i(1 + \gamma I_i).$$

Exercise 5.10. Code this scheme. What is the effect of varying dt?

Exercise 5.11. For the disease to become an epidemic, the number of newly infected has to be larger than the number of recovered. That is,

$$\lambda S_i I_i - \gamma I_i > 0 \Leftrightarrow S_i > \gamma/\lambda.$$

Can you observe this in your simulations?

The parameter γ has a simple interpretation. Suppose that a person stays ill for δ days before recovering. If I_t is relatively stable, that means every day the same number of people get infected as recover, and therefore a $1/\delta$ fraction of people recover each day. Thus, γ is the reciprocal of the duration of the infection in a given person.

Chapter 6

Google PageRank

6.1 Basic ideas

We are going to simulate the Internet. In particular, we are going to simulate the *Pagerank* algorithm by which *Google* determines the importance of web pages.

Let's start with some basic classes:

- A `Page` contains some information such as its title and a global numbering in Google's data-center. It also contains a collection of links.
- We represent a link with a pointer to a `Page`. Conceivably we could have a `Link` class, containing further information such as probability of being clicked, or number of times clicked, but for now a pointer will do.
- Ultimately we want to have a class `Web` which contains a number of pages and their links. The web object will ultimately also contain information such as relative importance of the pages.

This application is a natural one for using pointers. When you click on a link on a web page you go from looking at one page in your browser to looking at another. You could implement this by having a pointer to a page, and clicking updates the value of this pointer.

Exercise 6.1. Make a class `Page` which initially just contains the name of the page. Write a method to display the page. Since we will be using pointers quite a bit, let this be the intended code for testing:

```
auto homepage = make_shared<Page>("My Home Page");
cout << "Homepage has no links yet:" << '\n';
cout << homepage->as_string() << '\n';
```

Next, add links to the page. A link is a pointer to another page, and since there can be any number of them, you will need a `vector` of them. Write a method `click` that follows the link. Intended code:

```
auto utexas = make_shared<Page>("University Home Page");
homepage->add_link(utexas);
auto searchpage = make_shared<Page>("google");
homepage->add_link(searchpage);
cout << homepage->as_string() << '\n';
```

Exercise 6.2. Add some more links to your homepage. Write a method `random_click` for the `Page` class. Intended code:

```
for (int iclick=0; iclick<20; iclick++) {
```

```
    auto newpage = homepage->random_click();
    cout << "To: " << newpage->as_string() << '\n';
  }
```

How do you handle the case of a page without links?

6.2 Clicking around

Exercise 6.3. Now make a class `Web` which foremost contains a bunch (technically: a `vector`) of pages. Or rather: of pointers to pages. Since we don't want to build a whole internet by hand, let's have a method `create_random_links` which makes a random number of links to random pages. Intended code:

```
Web internet(netsize);
internet.create_random_links(avglinks);
```

Now we can start our simulation. Write a method `Web::random_walk` that takes a page, and the length of the walk, and simulates the result of randomly clicking that many times on the current page. (Current page. Not the starting page.)

Let's start working towards PageRank. First we see if there are pages that are more popular than others. You can do that by starting a random walk once on each page. Or maybe a couple of times.

Exercise 6.4. Apart from the size of your internet, what other design parameters are there for your tests? Can you give a back-of-the-envelope estimation of their effect?

Exercise 6.5. Your first simulation is to start on each page a number of times, and counts where that lands you. Intended code:

```
vector<int> landing_counts(internet.number_of_pages(),0);
for ( auto page : internet.all_pages() ) {
  for (int iwalk=0; iwalk<5; iwalk++) {
    auto endpage = internet.random_walk(page,2*avglinks,tracing);
    landing_counts.at(endpage->global_ID())++;
  }
}
```

Display the results and analyze. You may find that you finish on certain pages too many times. What's happening? Fix that.

6.3 Graph algorithms

There are many algorithms that rely on gradually traversing the web. For instance, any graph can be *connected*. You test that by

- Take an arbitrary vertex v. Make a 'reachable set' $R \leftarrow \{v\}$.
- Now see where you can get from your reachable set:

$$\forall_{v \in V} \forall_{w \text{ neighbour of } v} : R \leftarrow R \cup \{w\}$$

- Repeat the previous step until R does not change anymore.

After this algorithm concludes, is R equal to your set of vertices? If so, your graph is called (fully) connected. If not, your graph has multiple *connected components*.

Exercise 6.6. Code the above algorithm, keeping track of how many steps it takes to reach each vertex w. This is the *Single Source Shortest Path* algorithm (for unweighted graphs).

The *diameter* is defined as the maximal shortest path. Code this.

6.4 Page ranking

The Pagerank algorithm now asks, if you keep clicking randomly, what is the distribution of how likely you are to wind up on a certain page. The way we calculate that is with a probability distribution: we assign a probability to each page so that the sum of all probabilities is one. We start with a random distribution:

Code:

```
1 ProbabilityDistribution
2
      random_state(internet.number_of_pages()
3 random_state.set_random();
4 cout << "Initial distribution: " <<
      random_state.as_string() << '\n';
```

Output
[google] pdfsetup:

```
Initial distribution:
    0:0.00, 1:0.02, 2:0.07,
    3:0.05, 4:0.06, 5:0.08,
    6:0.04, 7:0.04, 8:0.04,
    9:0.01, 10:0.07, 11:0.05,
    12:0.01, 13:0.04,
    14:0.08, 15:0.06,
    16:0.10, 17:0.06,
    18:0.11, 19:0.01,
```

Exercise 6.7. Implement a class `ProbabilityDistribution`, which stores a vector of floating point numbers. Write methods for:

- accessing a specific element,
- setting the whole distribution to random, and
- normalizing so that the sum of the probabilities is 1.
- a method rendering the distribution as string could be useful too.

Next we need a method that given a probability distribution, gives you the new distribution corresponding to performing a single click. (This is related to *Markov chains*; see HPC book [6], section 9.2.1.)

Exercise 6.8. Write the method

```
ProbabilityDistribution Web::globalclick
    ( ProbabilityDistribution currentstate );
```

Test it by

- start with a distribution that is nonzero in exactly one page;
- print the new distribution corresponding to one click;
- do this for several pages and inspect the result visually.

Then start with a random distribution and run a couple of iterations. How fast does the process converge? Compare the result to the random walk exercise above.

Exercise 6.9. In the random walk exercise you had to deal with the fact that some pages have no outgoing links. In that case you transitioned to a random page. That mechanism is lacking in the `globalclick` method. Figure out a way to incorporate this.

Let's simulate some simple 'search engine optimization' trick.

Exercise 6.10. Add a page that you will artificially made look important: add a number of pagesthat all link to this page, but no one links to them. (Because of the random clicking they will still sometimes be reached.)

Compute the rank of the artificially hyped page. Did you manage to trick Google into ranking this page high? How many links did you have to add?

Sample output:

```
Internet has 5000 pages
Top score: 109:0.0013, 3179:0.0012, 4655:0.0010, 3465:0.0009, 4298:0.0008
With fake pages:
Internet has 5051 pages
Top score: 109:0.0013, 3179:0.0012, 4655:0.0010, 5050:0.0010, 4298:0.0008
Hyped page scores at 4
```

6.5 Graphs and linear algebra

The probability distribution is essentially a vector. You can also represent the web as a matrix W with $w_{ij} = 1$ if page i links to page j. How can you interpret the `globalclick` method in these terms?

Exercise 6.11. Add the matrix representation of the `Web` object and reimplement the `globalclick` method. Test for correctness.

Do a timing comparison.

The iteration you did above to find a stable probability distribution corresponds to the 'power method' in linear algebra. Look up the Perron-Frobenius theory and see what it implies for page ranking.

Chapter 7

Redistricting

In this project you can explore 'gerrymandering', the strategic drawing of districts to give a minority population a majority of districts[1].

7.1 Basic concepts

We are dealing with the following concepts:

- A state is divided into census districts, which are given. Based on census data (income, ethnicity, median age) one can usually make a good guess as to the overall voting in such a district.
- There is a predetermined number of congressional districts, each of which consists of census districts. A congressional district is not a random collection: the census districts have to be contiguous.
- Every couple of years, to account for changing populations, the district boundaries are redrawn. This is known as redistricting.

There is considerable freedom in how redistricting is done: by shifting the boundaries of the (congressional) districts it is possible to give a population that is in the overall minority a majority of districts. This is known as *gerrymandering*.

For background reading, see `https://redistrictingonline.org/`.

To do a small-scale computer simulation of gerrymandering, we make some simplifying assumption.

- First of all, we dispense with census district: we assume that a district consists directly of voters, and that we know their affiliation. In practice one relies on proxy measures (such as income and education level) to predict affiliation.
- Next, we assume a one-dimensional state. This is enough to construct examples that bring out the essence of the problem:

  Consider a state of five voters, and we designate their votes as `AAABB`. Assigning them to three (contiguous) districts can be done as `AAA|B|B`, which has one 'A' district and two 'B' districts.
- We also allow districts to be any positive size, as long as the number of districts is fixed.

1. This project is obviously based on the Northern American political system. Hopefully the explanations here are clear enough. Please contact the author if you know of other countries that have a similar system.

7.2 Basic functions

7.2.1 Voters

We dispense with census districts, expressing everything in terms of voters, for which we assume a known voting behavior. Hence, we need a `Voter` class, which will record the voter ID and party affiliation. We assume two parties, and leave an option for being undecided.

Exercise 7.1. Implement a `Voter` class. You could for instance let ± 1 stand for `A/B`, and 0 for undecided.

```
cout << "Voter 5 is positive:" << '\n';
Voter nr5(5,+1);
cout << nr5.print() << '\n';

cout << "Voter 6 is negative:" << '\n';
Voter nr6(6,-1);
cout << nr6.print() << '\n';

cout << "Voter 7 is weird:" << '\n';
Voter nr7(7,3);
cout << nr7.print() << '\n';
```

7.2.2 Populations

Exercise 7.2. Implement a `District` class that models a group of voters.

- You probably want to create a district out of a single voter, or a vector of them. Having a constructor that accepts a string representation would be nice too.
- Write methods `majority` to give the exact majority or minority, and `lean` that evaluates whether the district overall counts as A party or B party.
- Write a `sub` method to creates subsets.

  ```
  District District::sub(int first,int last);
  ```

- For debugging and reporting it may be a good idea to have a method

  ```
  string District::print();
  ```

Code:

```
    cout << "Making district with one B
    voter" << '\n';
    Voter nr5(5,+1);
    District nine( nr5 );
    cout << ".. size: " << nine.size() <<
    '\n';
    cout << ".. lean: " << nine.lean() <<
    '\n';
    /* ... */
    cout << "Making district ABA" << '\n';
    District nine( vector<Voter>
                   { {1,-1},{2,+1},{3,-1}
    } );
    cout << ".. size: " << nine.size() <<
    '\n';
    cout << ".. lean: " << nine.lean() <<
    '\n';
```

Output

```
[gerry] district:
Making district with one B voter
.. size: 1
.. lean: 1

Making district ABA
.. size: 3
.. lean: -1
```

Exercise 7.3. Implement a `Population` class that will initially model a whole state.

Code:

```
string pns( "-++--" );
Population some(pns);
cout << "Population from string " << pns
     << '\n';
cout << ".. size: " << some.size() <<
     '\n';
cout << ".. lean: " << some.lean() <<
     '\n';
Population group=some.sub(1,3);
cout << "sub population 1--3" << '\n';
cout << ".. size: " << group.size() <<
     '\n';
cout << ".. lean: " << group.lean() <<
     '\n';
```

Output

```
[gerry] population:
Population from string -++--
.. size: 5
.. lean: -1
sub population 1--3
.. size: 2
.. lean: 1
```

In addition to an explicit creation, also write a constructor that specifies how many people and what the majority is:

```
Population( int population_size,int majority,bool trace=false )
```

Use a random number generator to achieve precisely the indicated majority.

7.2.3 Districting

The next level of complication is to have a set of districts. Since we will be creating this incrementally, we need some methods for extending it.

Exercise 7.4. Write a class `Districting` that stores a `vector` of `District` objects. Write `size` and `lean` methods:

Code:

```
cout << "Making single voter population
    B" << '\n';
Population people( vector<Voter>{
    Voter(0,+1) } );
cout << ".. size: " << people.size() <<
    '\n';
cout << ".. lean: " << people.lean() <<
    '\n';

Districting gerry;
cout << "Start with empty districting:"
    << '\n';
cout << ".. number of districts: " <<
    gerry.size() << '\n';
```

Output

```
[gerry] gerryempty:
Making single voter population B
.. size: 1
.. lean: 1
Start with empty districting:
.. number of districts: 0
```

Exercise 7.5. Write methods to extend a `Districting`:

```
cout << "Add one B voter:" << '\n';
gerry = gerry.extend_with_new_district( people.at(0) );
cout << ".. number of districts: " << gerry.size() << '\n';
cout << ".. lean: " << gerry.lean() << '\n';
cout << "add A A:" << '\n';
gerry = gerry.extend_last_district( Voter(1,-1) );
gerry = gerry.extend_last_district( Voter(2,-1) );
cout << ".. number of districts: " << gerry.size() << '\n';
cout << ".. lean: " << gerry.lean() << '\n';

cout << "Add two B districts:" << '\n';
gerry = gerry.extend_with_new_district( Voter(3,+1) );
gerry = gerry.extend_with_new_district( Voter(4,+1) );
cout << ".. number of districts: " << gerry.size() << '\n';
cout << ".. lean: " << gerry.lean() << '\n';
```

7.3 Strategy

Now we need a method for districting a population:

```
Districting Population::minority_rules( int ndistricts );
```

Rather than generating all possible partitions of the population, we take an incremental approach (this is related to the solution strategy called *dynamic programming*):

- The basic question is to divide a population optimally over n districts;
- We do this recursively by first solving a division of a subpopulation over $n-1$ districts,
- and extending that with the remaining population as one district.

This means that you need to consider all the ways of having the 'remaining' population into one district, and that means that you will have a loop over all ways of splitting the population, outside of your recursion; see figure 51.1 (textbook).

- For all $p = 0, \ldots n-1$ considering splitting the state into $0, \ldots, p-1$ and $p, \ldots, n-1$.
- Use the best districting of the first group, and make the last group into a single district.

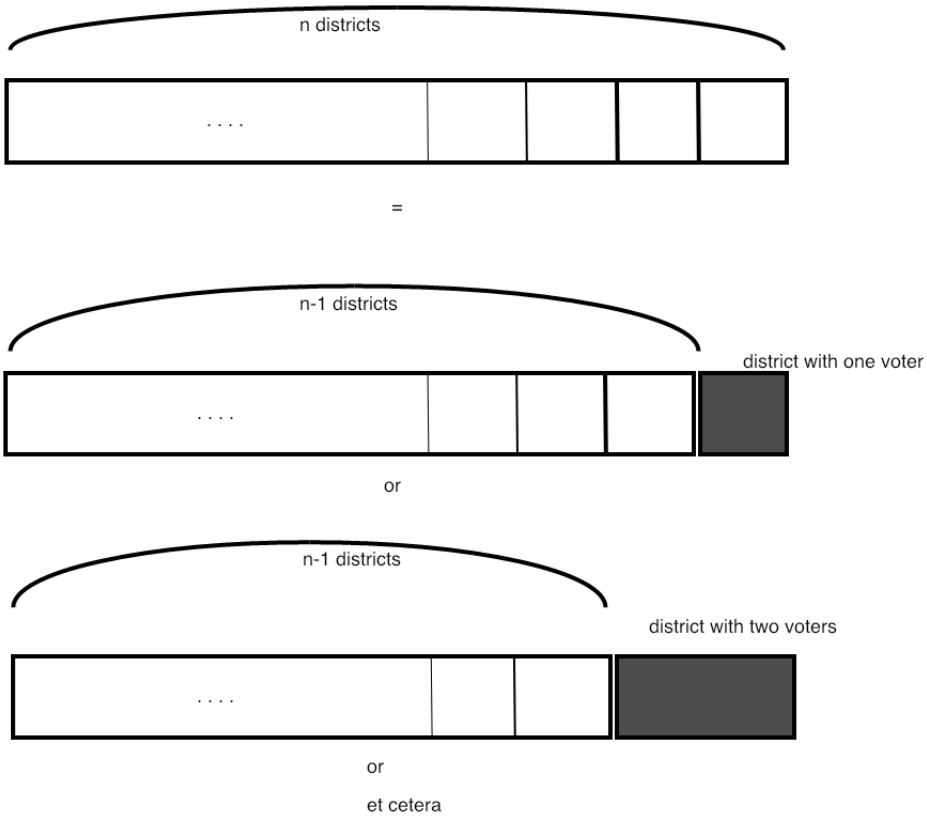


Figure 7.1: Multiple ways of splitting a population

- Keep the districting that gives the strongest minority rule, over all values of p.

You can now realize the above simple example:

```
AAABB => AAA|B|B
```

Exercise 7.6. Implement the above scheme.

Code:

```
1  Population five("+++--");
2  cout << "Redistricting population: " <<
       '\n'
3       << five.print() << '\n';
4  cout << ".. majority rule: "
5       << five.rule() << '\n';
6  int ndistricts{3};
7  auto gerry =
       five.minority_rules(ndistricts);
8  cout << gerry.print() << '\n';
9  cout << ".. minority rule: "
10      << gerry.rule() << '\n';
```

Output

```
[gerry] district5:

Redistricting population:
[0:+,1:+,2:+,3:-,4:-,]
.. majority rule: 1
[3[0:+,1:+,2:+,],[3:-,],[4:-,],]
.. minority rule: -1
```

Note: the range for p given above is not quite correct: for instance, the initial part of the population needs to be big enough to accommodate $n - 1$ voters.

Exercise 7.7. Test multiple population sizes; how much majority can you give party B while still giving party A a majority.

7.4 Efficiency: dynamic programming

If you think about the algorithm you just implemented, you may notice that the districtings of the initial parts get recomputed quite a bit. A strategy for optimizing for this is called *memoization*.

Exercise 7.8. Improve your implementation by storing and reusing results for the initial sub-populations.

In a way, we solved the program backward: we looked at making a district out of the last so-many voters, and then recursively solving a smaller problem for the first however-many voters. But in that process, we decided what is the best way to assign districts to the first 1 voter, first 2, first 3, et cetera. Actually, for more than one voter, say five voters, we found the result on the best attainable minority rule assigning these five voters to one, two, three, four districts.

The process of computing the 'best' districting forward, is known as *dynamic programming*. The fundamental assumption here is that you can use intermediate results and extend them, without having to reconsider the earlier problems.

Consider for instance that you've considered districting ten voters over up to five districts. Now the majority for eleven voters and five districts is the minimum of

- ten voters and five districts, and the new voter is added to the last district; or
- ten voters and four districts, and the new voter becomes a new district.

Exercise 7.9. Code a dynamic programming solution to the redistricting problem.

7.5 Extensions

The project so far has several simplifying assumptions.

- Congressional districts need to be approximately the same size. Can you put a limit on the ratio between sizes? Can the minority still gain a majority?

Exercise 7.10. The biggest assumption is of course that we considered a one-dimensional state. With two dimensions you have more degrees of freedom of shaping the districts. Implement a two-dimensional scheme; use a completely square state, where the census districts form a regular grid. Limit the shape of the congressional districts to be convex.

The *efficiency gap* is a measure of how 'fair' a districting of a state is.

Exercise 7.11. Look up the definition of efficiency gap (and 'wasted votes'), and implement it in your code.

7.6 Ethics

The activity of redistricting was intended to give people a fair representation. In its degenerate form of Gerrymandering this concept of fairness is violated because the explicit goal is to give the minority a majority of votes. Explore ways that this unfairness can be undone.

In your explorations above, the only characteristic of a voter was their preference for party A or B. However, in practice voters can be considered part of communities. The Voting Rights Act is concerned about 'minority vote dilution'. Can you show examples that a color-blind districting would affect some communities negatively?

Chapter 8

Amazon delivery truck scheduling

This section contains a sequence of exercises that builds up to a simulation of delivery truck scheduling.

8.1 Problem statement

Scheduling the route of a delivery truck is a well-studied problem. For instance, minimizing the total distance that the truck has to travel corresponds to the *Traveling Salesman Problem (TSP)*. However, in the case of *Amazon delivery truck* scheduling the problem has some new aspects:

- A customer is promised a window of days when delivery can take place. Thus, the truck can split the list of places into sublists, with a shorter total distance than going through the list in one sweep.
- Except that *Amazon prime* customers need their deliveries guaranteed the next day.

8.2 Coding up the basics

Before we try finding the best route, let's put the basics in place to have any sort of route at all.

8.2.1 Address list

You probably need a class `Address` that describes the location of a house where a delivery has to be made.

- For simplicity, let give a house (i, j) coordinates.
- We probably need a `distance` function between two addresses. We can either assume that we can travel in a straight line between two houses, or that the city is build on a grid, and you can apply the so-called *Manhattan distance*.
- The address may also require a field recording the last possible delivery date.

Exercise 8.1. Code a class `Address` with the above functionality, and test it.

Code:

```
1     Address one(1.,1.),
2       two(2.,2.);
3     cerr << "Distance: "
4          << one.distance(two)
5          << '\n';
```

Output

```
[amazon] address:

Address
Distance: 1.41421
.. address
Address 1 should be closest to
    the depot. Check: 1

Route from depot to depot:
    (0,0) (2,0) (1,0) (3,0)
    (0,0)
 has length 8: 8
Greedy scheduling:  (0,0) (1,0)
    (2,0) (3,0) (0,0)
 should have length 6: 6

Square5
Travel in order: 24.1421
Square route:  (0,0) (0,5)
    (5,5) (5,0) (0,0)
 has length 20
.. square5

Original list:  (0,0) (-2,0)
    (-1,0) (1,0) (2,0) (0,0)
 length=8
flip middle two addresses:
    (0,0) (-2,0) (1,0) (-1,0)
    (2,0) (0,0)
 length=12
better:  (0,0) (1,0) (-2,0)
    (-1,0) (2,0) (0,0)
 length=10

Hundred houses
Route in order has length
    25852.6
TSP based on mere listing has
    length: 2751.99 over naive
    25852.6
Single route has length: 2078.43
 .. new route accepted with
    length 2076.65
Final route has length 2076.65
    over initial 2078.43
TSP route has length 1899.4
    over initial 2078.43

Two routes
Route1:  (0,0) (2,0) (3,2)
    (2,3) (0,2) (0,0)
route2:  (0,0) (3,1) (2,1)
    (1,2) (1,3) (0,0)
total length 19.6251
start with 9.88635,9.73877
Pass 0
.. down to 9.81256,8.57649
Pass 1
Pass 2
Pass 3
Pass 4
TSP Route1:  (0,0) (3,1) (3,2)
    (2,3) (0,2) (0,0)
route2:  (0,0) (2,0) (2,1)
    (1,2) (1,3) (0,0)
total length 18.389
```

Next we need a class `AddressList` that contains a list of addresses.

Exercise 8.2. Implement a class `AddressList`; it probably needs the following methods:

- `add_address` for constructing the list;
- `length` to give the distance one has to travel to visit all addresses in order;
- `index_closest_to` that gives you the address on the list closest to another address, presumably not on the list.

8.2.2 Add a depot

Next, we model the fact that the route needs to start and end at the depot, which we put arbitrarily at coordinates $(0, 0)$. We could construct an `AddressList` that has the depot as first and last element, but that may run into problems:

- If we reorder the list to minimize the driving distance, the first and last elements may not stay in place.
- We may want elements of a list to be unique: having an address twice means two deliveries at the same address, so the `add_address` method would check that an address is not already in the list.

We can solve this by making a class `Route`, which inherits from `AddressList`, but the methods of which leave the first and last element of the list in place.

8.2.3 Greedy construction of a route

Next we need to construct a route. Rather than solving the full TSP, we start by employing a *greedy search* strategy:

> Given a point, find the next point by some local optimality test, such as shortest distance. Never look back to revisit the route you have constructed so far.

Such a strategy is likely to give an improvement, but most likely will not give the optimal route.

Let's write a method

```
Route::Route greedy_route();
```

that constructs a new address list, containing the same addresses, but arranged to give a shorter length to travel.

Exercise 8.3. Write the `greedy_route` method for the `AddressList` class.

1. Assume that the route starts at the depot, which is located at $(0, 0)$. Then incrementally construct a new list by:
2. Maintain an `Address` variable `we_are_here` of the current location;
3. repeatedly find the address closest to `we_are_here`.

Extend this to a method for the `Route` class by working on the subvector that does not contain the final element.

Test it on this example:

Code:

```
1   Route deliveries;
2   deliveries.add_address( Address(0,5)
    );
3   deliveries.add_address( Address(5,0)
    );
4   deliveries.add_address( Address(5,5)
    );
5   cerr << "Travel in order: " <<
    deliveries.length() << '\n';
6   assert( deliveries.size()==5 );
7   auto route =
    deliveries.greedy_route();
8   assert( route.size()==5 );
9   auto len = route.length();
10  cerr << "Square route: " <<
    route.as_string()
11        << "\n has length " << len <<
    '\n';
```

Output

```
[amazon] square5:

Travel in order: 24.1421
Square route:  (0,0) (0,5)
    (5,5) (5,0) (0,0)
 has length 20
```

Reorganizing a list can be done in a number of ways.

- First of all, you can try to make the changes in place. This runs into the objection that maybe you want to save the original list; also, while swapping two elements can be done with the `insert` and `erase` methods, more complicated operations are tricky.
- Alternatively, you can incrementally construct a new list. Now the main problem is to keep track of which elements of the original have been processed. You could do this by giving each address a boolean field `done`, but you could also make a copy of the input list, and remove the elements that have been processed. For this, study the `erase` method for `vector` objects.

8.3 Optimizing the route

The above suggestion of each time finding the closest address is known as a *greedy search* strategy. It does not give you the optimal solution of the TSP. Finding the optimal solution of the TSP is hard to program – you could do it recursively – and takes a lot of time as the number of addresses grows. In fact, the TSP is probably the most famous of the class of *NP-hard* problems, which are generally believed to have a running time that grows faster than polynomial in the problem size.

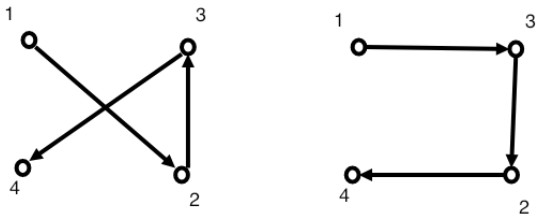


Figure 8.1: Illustration of the 'opt2' idea of reversing part of a path

However, you can approximate the solution heuristically. One method, the Kernighan-Lin algorithm [12], is based on the *opt2* idea: if you have a path that 'crosses itself', you can make it shorter by reversing

part of it. Figure 52.1 (textbook) shows that the path $1--2--3--4$ can be made shorter by reversing part of it, giving $1--3--2--4$. Since recognizing where a path crosses itself can be hard, or even impossible for graphs that don't have Cartesian coordinates associated, we adopt a scheme where we try all possible reversals:

```
for all nodes m<n on the path [1..N]:
  make a new route from
    [1..m-1] + [m--n].reversed + [n+1..N]
  if the new route is shorter, keep it
```

Exercise 8.4. Code the opt2 heuristic: write a method to reverse part of the route, and write the loop that tries this with multiple starting and ending points. Try it out on some simple test cases to convince you that your code works as intended.

Let's explore issues of complexity. (For an introduction to complexity calculations, see HPC book [6], section 14.) The TSP is one of a class of *NP complete* problems, which very informally means that there is no better solution than trying out all possibilities.

Exercise 8.5. What is the runtime complexity of the heuristic solution using opt2? What would the runtime complexity be of finding the best solution by considering all possibilities? Make a very rough estimation of runtimes of the two strategies on some problem sizes: $N = 10, 100, 1000, \ldots$.

Exercise 8.6. Earlier you had programmed the greedy heuristic. Compare the improvement you get from the opt2 heuristic, starting both with the given list of addresses, and with a greedy traversal of it.

For realism, how many addresses do you put on your route? How many addresses would a delivery driver do on a typical day?

8.4 Multiple trucks

If we introduce multiple delivery trucks, we get the 'Multiple Traveling Salesman Problem' [3]. With this we can module both the cases of multiple trucks being out on delivery on the same day, or one truck spreading deliveries over multiple days. For now we don't distinguish between the two.

The first question is how to divide up the addresses.

1. We could split the list in two, using some geometric test. This is a good model for the case where multiple trucks are out on the same day. However, if we use this as a model for the same truck being out on multiple days, we are missing the fact that new addresses can be added on the first day, messing up the neatly separated routes.
2. Thus it may in fact be reasonable to assume that all trucks get an essentially random list of addresses.

Can we extend the opt2 heuristic to the case of multiple paths? For inspiration take a look at figure 52.2 (textbook): instead of modifying one path, we could switch bits out bits between one path and another. When you write the code, take into account that the other path may be running backwards! This means that based on split points in the first and second path you know have four resulting modified paths to consider.

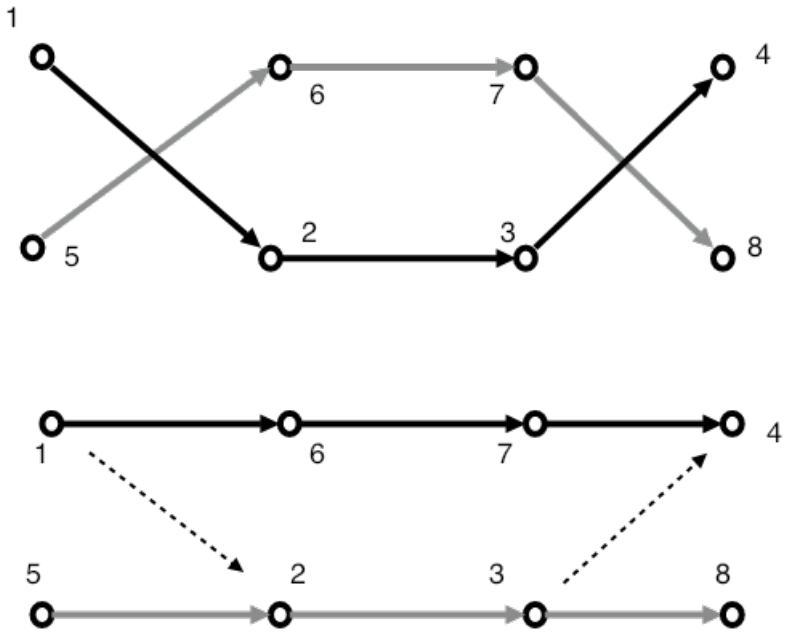


Figure 8.2: Extending the 'opt2' idea to multiple paths

Exercise 8.7. Write a function that optimizes two paths simultaneously using the multi-path version of the opt2 heuristic. For a test case, see figure 52.3 (textbook).

You have quite a bit of freedom here:

- The start points of the two segments should be chosen independently;
- the lengths can be chosen independently, but need not; and finally
- each segment can be reversed.

More flexibility also means a longer runtime of your program. Does it pay off? Do some tests and report results.

Based on the above description there will be a lot of code duplication. Make sure to introduce functions and methods for various operations.

8.5 Amazon prime

In section 52.4 (textbook) you made the assumption that it doesn't matter on what day a package is delivered. This changes with *Amazon prime*, where a package has to be delivered guaranteed on the next day.

Exercise 8.8. Explore a scenario where there are two trucks, and each have a number of addresses that can not be exchanged with the other route. How much longer is the total distance? Experiment with the ratio of prime to non-prime addresses.

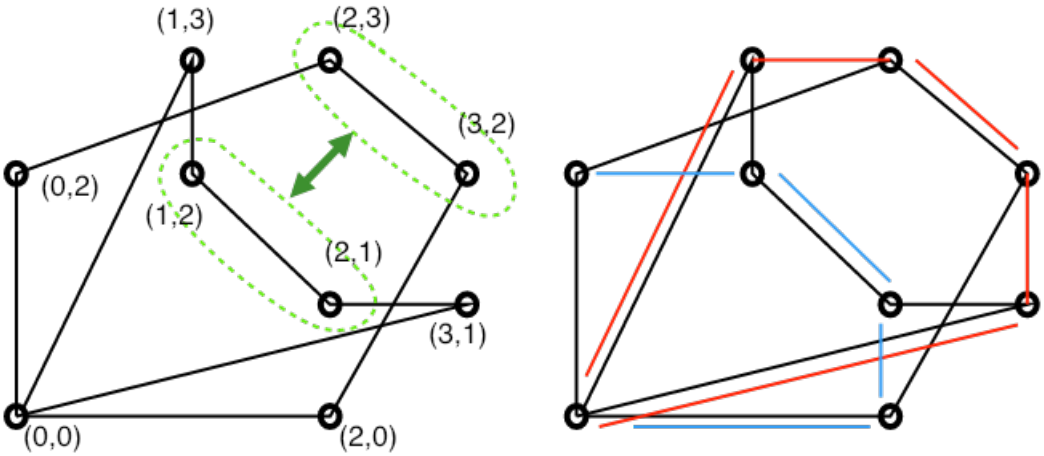


Figure 8.3: Multiple paths test case

8.6 Dynamicism

So far we have assumed that the list of addresses to be delivered to is given. This is of course not true: new deliveries will need to be scheduled continuously.

Exercise 8.9. Implement a scenario where every day a random number of new deliveries is added to the list. Explore strategies and design choices.

8.7 Ethics

People sometimes criticize Amazon's labor policies, including regarding its drivers. Can you make any observations from your simulations in this respect?

Chapter 9

The Great Garbage Patch

This section contains a sequence of exercises that builds up to a *cellular automaton* simulation of turtles in the ocean, garbage that is deadly to them, and ships that clean it up. To read more about this: https://theoceancleanup.com/.

Thanks to Ernesto Lima of TACC for the idea and initial code of this exercise.

9.1 Problem and model solution

There is lots of plastic floating around in the ocean, and that is harmful to fish and turtles and cetaceans. Here you get to model the interaction between

- Plastic, randomly located;
- Turtles that swim about; they breed slowly, and they die from ingesting plastic;
- Ships that sweep the ocean to remove plastic debris.

The simulation method we use is that of a *cellular automaton*:

- We have a grid of cells;
- Each cell has a 'state' associated with it, namely, it can contain a ship, a turtle, or plastic, or be empty; and
- On each next time step, the state of a cell is a simple function of the states of that cell and its immediate neighbors.

The purpose of this exercise is to simulate a number of time steps, and explore the iteraction between parameters: with how much gargage will turtles die out, how many ships are enough to protect the turtles.

9.2 Program design

The basic idea is to have an `ocean` object that is populated with turtles, trash, and ships. Your simulation will let the ocean undergo a number of time steps:

```
for (int t=0; t<time_steps; t++)
  ocean.update();
```

Ultimately your purpose is to investigate the development of the turtle population: is it stable, does it die out?

While you can make a 'hackish' solution to this problem, partly you will be judged on your use of modern/clean C++ programming techniques. A number of suggestions are made below.

9.2.1 Grid update

Here is a point to be aware of. Can you see what's wrong with with doing an update entirely in-place:

```
for ( i )
  for ( j )
    cell(i,j) = f( cell(i,j), .... other cells ... );
```

?

9.3 Testing

It can be complicated to test this program for correctness. The best you can do is to try out a number of scenarios. For that it's best to make your program input flexible: use the `cxxopts` package [4], and drive your program from a shell script.

Here is a list of things you can test.

1. Start with only a number of ships; check that after 1000 time steps you still have the same number.
2. Likewise with turtles; if they don't breed and don't die, check that their number stays constant.
3. With only ships and trash, does it all get swept?
4. With only turtles and trash, do they all die off?

It is harder to test that your turtles and ships don't 'teleport' around, but only move to contiguous cells. For that, use visual inspection; see section 9.3.1.

9.3.1 Animated graphics

The output of this program is a prime candidate for visualization. In fact, some test ('make sure that turtles don't teleport') are hard to do other than by looking at the output. Start by making an ascii rendering of the ocean grid, as in figure 9.1.

It would be better to have some sort of animated output. However, not all programming languages generate visual output equally easily. There are very powerful video/graphics libraries in C++, but these are also hard to use. There is a simpler way out.

For simple output such as this program yields, you can make a simple low-budget animation. Every *terminal emulator* under the sun supports *VT100 cursor control* [1]: you can send certain magic output to your screen to control cursor positioning.

In each time step you would

1. Send the cursor to the home position, by this magic output:

```
#include <cstdio>
    /* ... */
    // ESC [ i ; j H
    printf( "%c[0;0H",(char)27);
```

2. Display your grid as in figure 9.1;
3. Sleep for a fraction of a second; see section 25.1.4 (textbook).

1. https://vt100.net/docs/vt100-ug/chapter3.html

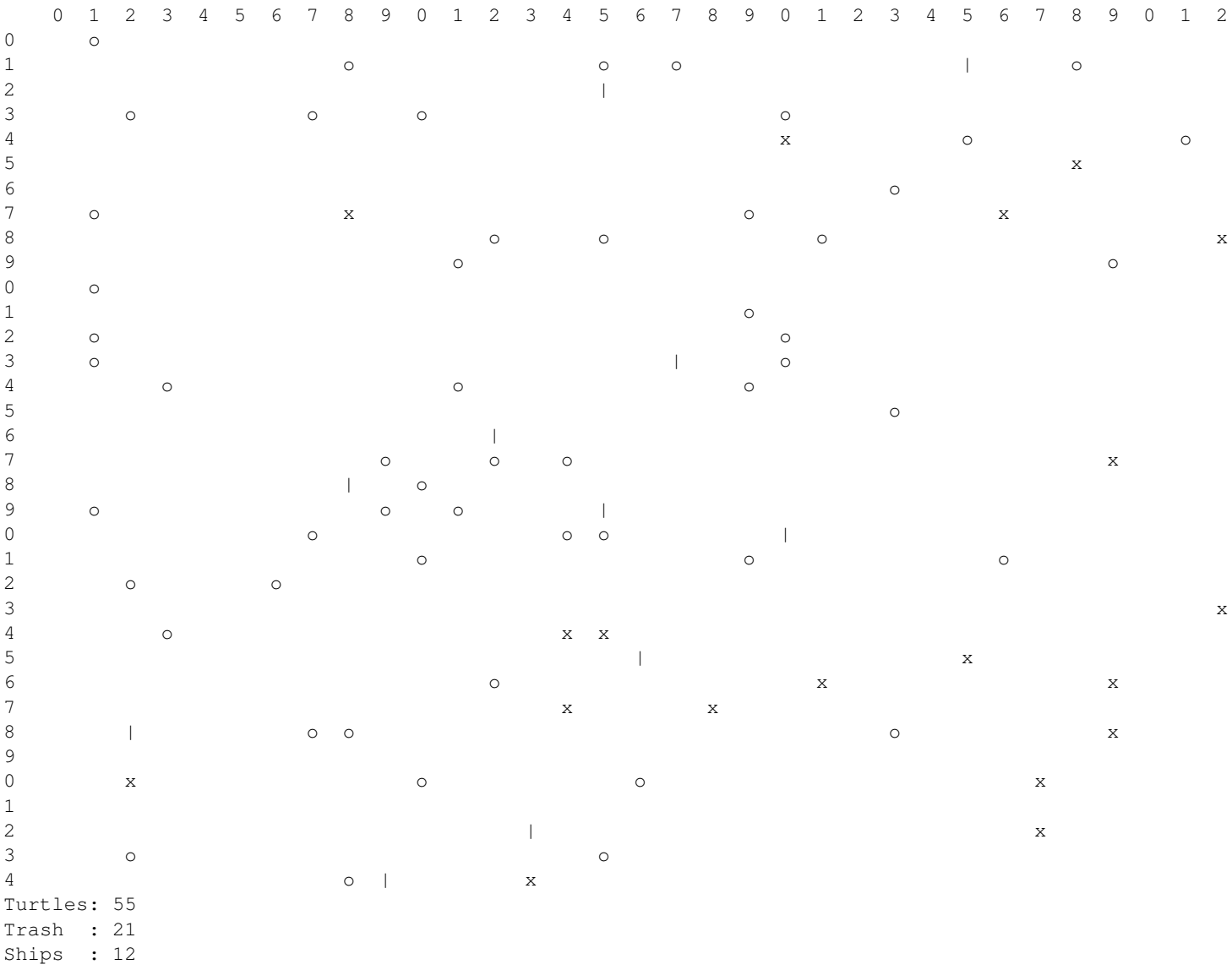


Figure 9.1: Ascii art printout of a time step

9.4 Modern programming techniques

9.4.1 Object oriented programming

While there is only have one ocean, you should still make an `ocean` class, rather than having a global array object. All functions are then methods of the single object you create of that class.

9.4.2 Data structure

We lose very little generality by ignoring the depth of the ocean and the shape of coastlines, and model the ocean as a 2D grid.

If you write an indexing function `cell(i,j)` you can make your code largely independent of the actual data structure chosen. Argue why `vector<vector<int>>` is not the best storage.

1. What do you use instead?
2. When you have a working code, can you show by timing that your choice is indeed superior?

9.4.3 Cell types

Having 'magic numbers' in your code (0 =empty, 1 =turtle, et cetera) is not elegant. Make a **`enum`** or **`enum class`** (see section 24.10 (textbook)) so that you have names for what's in your cells:

```
cell(i,j) = occupy::turtle;
```

If you want to print out your ocean, it might be nice if you can directly **`cout`** your cells:

```
for (int i=0; i<iSize; i++) {
  cout << i%10 << " ";
  for (int j=0; j<jSize; j++) {
    cout << setw(hs) << cell(i,j);
  }
  cout << '\n';
}
```

9.4.4 Ranging over the ocean

It is easy enough to write a loop as

```
for (int i=0; i<iSize; i++)
  for (int j=0; j<jSize; j++)
    ... cell(i,j) ...
```

However, it may not be a good idea to always sweep over your domain so orderly. Can you implement this:

```
for ( auto [i,j] : permuted_indices() ) {
    ... cell(i,j) ...
```

? See section 24.5 (textbook) about *structured binding*.

Likewise, if you need to count how many pieces of trash there are around a turtle, can you get this code to work:

```
int count_around( int ic,int jc,occupy typ ) const {
  int count=0;
  for ( auto [i,j] : neighbors(ic,jc) ) {
    if (cell(i,j) == typ)
      count++;
  }
  return count;
};
```

9.4.5 Random numbers

For the random movement of ships and turtles you need a random number generator. Do not use the old C generator, but the new `random` one; section 24.7 (textbook).

Try to find a solution so that you use exactly one generator for all places where you need random numbers. Hint: make the generator `static` in your class.

9.5 Explorations

Instead of having the ships move randomly, can you give them a preferential direction to the closest garbage patch? Does this improve the health of the turtle population?

Can you account for the relative size of ships and turtles by having a ship occupy a 2×2 block in your grid?

So far you have let trash stay in place. What if there are ocean currents? Can you make the trash 'sticky' so that trash particles start moving as a patch if they touch?

Turtles eat sardines. (No they don't.) What happens to the sardine population if turtles die out? Can you come up with parameter values that correspond to a stable ecology or a de-stabilized one?

9.5.1 Code efficiency

Investigate whether your implementation of the `enum` in section 9.4.3 has any effect on timing. Parse the fine print of section 24.10 (textbook).

You may remark that ranging over a largely empty ocean can be pretty inefficient. You could contemplate keeping an 'active list' of where the turtles et cetera are located, and only looping over that. How would you implement that? Do you expect to see a difference in timing? Do you actually?

How is runtime affected by choosing a vector-of-vectors implementation for the ocean; see section 9.4.2.

Chapter 10

High performance linear algebra

Linear algebra is fundamental to much of computational science. Applications involving Partial Diffential Equations (PDEs) come down to solving large systems of linear equation; solid state physics involves large eigenvalue systems. But even outside of engineering applications linear algebra is important: the major computational part of Deep Learning (DL) networks involves matrix-matrix multiplications.

Linear algebra operations such as the matrix-matrix product are easy to code in a naive way. However, this does not lead to high performance. In these exercises you will explore the basics of a strategy for high performance.

10.1 Mathematical preliminaries

The matrix-matrix product $C \leftarrow A \cdot B$ is defined as

$$\forall_{ij} : c_{ij} \leftarrow \sum_k a_{ik} b_{kj}.$$

Straightforward code for this would be:

```
for (i=0; i<a.m; i++)
  for (j=0; j<b.n; j++)
    s = 0;
    for (k=0; k<a.n; k++)
      s += a[i,k] * b[k,j];
    c[i,j] = s;
```

However, this is not the only way to code this operation. The loops can be permuted, giving a total of six implementations.

Exercise 10.1. Code one of the permuted algorithms and test its correctness. If the reference algorithm above can be said to be 'inner-product based', how would you describe your variant?

Yet another implementation is based on a block partitioning. Let A, B, C be split on 2×2 block form:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then

$$\begin{aligned} C_{11}&=A_{11}B_{11}+A_{12}B_{21},\\ C_{12}&=A_{11}B_{12}+A_{12}B_{22},\\ C_{21}&=A_{21}B_{11}+A_{22}B_{21},\\ C_{22}&=A_{21}B_{12}+A_{22}B_{22} \end{aligned} \tag{10.1}$$

Convince yourself that this actually computes the same product $C = A \cdot B$. For more on block algorithms, see HPC book [6], section 5.3.6.

Exercise 10.2. Write a matrix class with a multiplication routine:

```
Matrix Matrix::MatMult( Matrix other );
```

First implement a traditional matrix-matrix multiplication, then make it recursive. For the recursive algorithm you need to implement sub-matrix handling: you need to extract submatrices, and write a submatrix back into the surrounding matrix.

10.2 Matrix storage

The simplest way to store an $M \times N$ matrix is as an array of length MN. Inside this array we can decide to store the rows end-to-end, or the columns. While this decision is obviously of practical importance for a library, from a point of performance it makes no difference.

Remark 5 *Historically, linear algebra software such as the Basic Linear Algebra Subprograms (BLAS) has used columnwise storage, meaning that the location of an element (i, j) is computed as $i + j \cdot M$ (we will use zero-based indexing throughout this project, both for code and mathematical expressions.) The reason for this stems from the origins of the BLAS in the Fortran language, which uses column-major ordering of array elements. On the other hand, static arrays (such as `x[5][6][7]`) in the C/C++ languages have row-major ordering, where element (i, j) is stored in location $j + i \cdot N$.*

Above, you saw the idea of block algorithms, which requires taking submatrices. For efficiency, we don't want to copy elements into a new array, so we want the submatrix to correspond to a subarray.

Now we have a problem: only a submatrix that consists of a sequence of columns is contiguous. The formula $i + j \cdot M$ for location of element (i, j) is no long correct if the matrix is a subblock of a larger matrix.

For this reason, linear algebra software describes a submatrix by three parameters M, N, LDA, where 'LDA' stands for 'leading dimension of A' (see BLAS [10], and Lapack [1]). This is illustrated in figure 53.1 (textbook).

Exercise 10.3. In terms of M, N, LDA, what is the location of the (i, j) element?

Implementationwise we also have a problem. If we use `std::vector` for storage, it is not possible to take subarrays, since C++ insists that a vector has its own storage. The solution is to use `span`; section 10.9.5 (textbook).

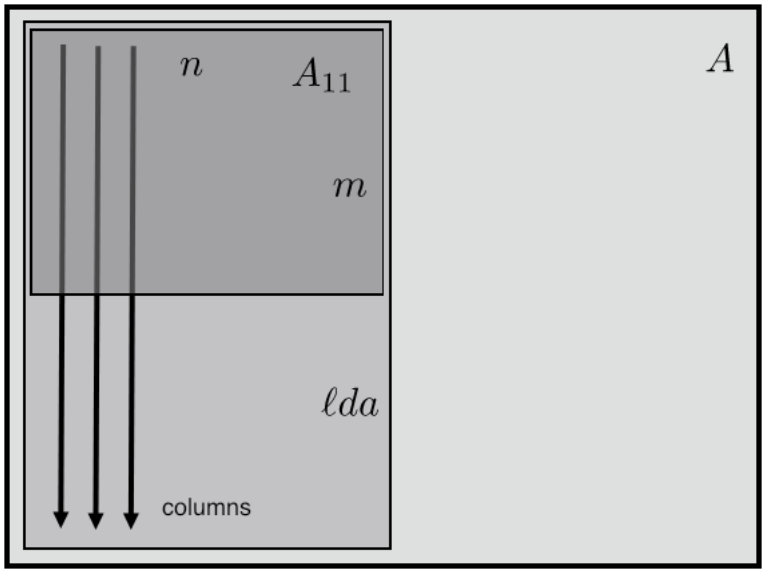


Figure 10.1: Submatrix out of a matrix, with M, N, LDA of the submatrix indicated

We could have two types of matrices: top level matrices that store a `vector<double>`, and submatrices that store a `span<double>`, but that is a lot of complication. It could be done using `std::variant` (section 24.6.4 (textbook)), but let's not.

Instead, let's adopt the following idiom, where we create a vector at the top level, and then create matrices from its memory.

```
// example values for M,LDA,N
M = 2; LDA = M+2; N = 3;
// create a vector to contain the data
vector<double> one_data(LDA*N,1.);
// create a matrix using the vector data
Matrix one(M,LDA,N,one_data.data());
```

(If you have not previously programmed in C, you need to get used to the `double*` mechanism. See section 10.10 (textbook).)

Exercise 10.4. Start implementing the `Matrix` class with a constructor

```
Matrix::Matrix(int m,int lda,int n,double *data)
```

and private data members:

```
private:
  int m,n,lda;
  span<double> data;
```

Write a method

```
double& Matrix::at(int i,int j);
```

that you can use as a safe way of accessing elements.

Let's start with simple operations.

Exercise 10.5. Write a method for adding matrices. Test it on matrices that have the same M, N, but different LDA.

Use of the `at` method is great for debugging, but it is not efficient. Use the preprocessor (chapter 21 (textbook)) to introduce alternatives:

```
#ifdef DEBUG
  c.at(i,j) += a.at(i,k) * b.at(k,j)
#else
  cdata[ /* expression with i,j */ ] += adata[ ... ] * bdata[ ... ]
#endif
```

where you access the data directly with

```
auto get_double_data() {
  double *adata;
  adata = data.data();
  return adata;
};
```

Exercise 10.6. Implement this. Use a cpp `#define` macro for the optimized indexing expression. (See section 21.2.2 (textbook).)

10.2.1 Submatrices

Next we need to support constructing actual submatrices. Since we will mostly aim for decomposition in 2×2 block form, it is enough to write four methods:

```
Matrix Left(int j);
Matrix Right(int j);
Matrix Top(int i);
Matrix Bot(int i);
```

where, for instance, `Left(5)` gives the columns with $j < 5$.

Exercise 10.7. Implement these methods and test them.

10.3 Multiplication

You can now write a first multiplication routine, for instance with a prototype

```
void Matrix::MatMult( Matrix& other,Matrix& out );
```

Alternatively, you could write

```
Matrix Matrix::MatMult( Matrix& other );
```

but we want to keep the amount of creation/destruction of objects to a minimum.

10.3.1 One level of blocking

Next, write

```
void Matrix::BlockedMatMult( Matrix& other,Matrix& out );
```

which uses the 2×2 form above.

10.3.2 Recursive blocking

The final step is to make the blocking recursive.

Exercise 10.8. Write a method

```
void RecursiveMatMult( Matrix& other,Matrix& out );
```

which

- Executes the 2×2 block product, using again `RecursiveMatMult` for the blocks.
- When the block is small enough, use the regular `MatMult` product.

10.4 Performance issues

If you experiment a little with the cutoff between the regular and recursive matrix-matrix product, you see that you can get good factor of performance improvement. Why is this?

The matrix-matrix product is a basic operation in scientific computations, and much effort has been put into optimizing it. One interesting fact is that it is just about the most optimizable operation under the sum. The reason for this is, in a nutshell, that it involves $O(N^3)$ operations on $O(N^2)$ data. This means that, in principle each element fetched will be used multiple times, thereby overcoming the *memory bottleneck*.

To understand performance issues relating to hardware, you need to do some reading. Section HPC book [6], section 1.3.4 explains the crucial concept of a *cache*.

Exercise 10.9. Argue that the naive matrix-matrix product implementation is unlikely actually to reuse data.

Explain why the recursive strategy does lead to data reuse.

Above, you set a cutoff point for when to switch from the recursive to the regular product.

Exercise 10.10. Argue that continuing to recurse will not have much benefit once the product is contained in the cache. What are the cache sizes of your processor?

Do experiments with various cutoff points. Can you relate this to the cache sizes?

10.4.1 Parallelism (optional)

The four clauses of equation 53.1 (textbook) target independent areas in the C matrix, so they could be executed in parallel on any processor that has at least four cores.

Explore the OpenMP library to parallelize the `BlockedMatMult`.

10.4.2 Comparison (optional)

The final question is: how close are you getting to the best possible speed? Unfortunately you are still a way off. You can explore that as follows.

Your computer is likely to have an optimized implementation, accessible through:

```
#include <cblas.h>

cblas_dgemm
  ( CblasColMajor, CblasNoTrans, CblasNoTrans,
    m,other.n,n, alpha,adata,lda,
    bdata,other.lda,
    beta,cdata,out.lda);
```

which computes $C \leftarrow \alpha A \cdot B + \beta C$.

Exercise 10.11. Use another cpp conditional to implement `MatMult` through a call to `cblas_dgemm`. What performance do you now get?

You see that your recursive implementation is faster than the naive one, but not nearly as fast as the CBlas one. This is because

- the CBlas implementation is probably based on an entirely different strategy [8], and
- it probably involves a certain amount of assembly coding.

Chapter 11

Graph algorithms

In this project you will explore some common graph algorithms, and their various possible implementations. The main theme here will be that the common textbook exposition of algorithms is not necessarily the best way to phrase them computationally.

As background knowledge for this project, you are encouraged to read HPC book [6], chapter 9; for an elementary tutorial on graphs, see HPC book [6], chapter 19.

11.1 Traditional algorithms

We first implement the 'textbook' formulations of two *Single Source Shortest Path (SSSP)* algorithms: on unweighted and then on weighted graphs. In the next section we will then consider formulations that are in terms of linear algebra.

In order to develop the implementations, we start with some necessary preliminaries,

11.1.1 Code preliminaries

11.1.1.1 Adjacency graph

We need a class `Dag` for a Directed Acyclic Graph (DAG):

```
class Dag {
private:
  vector< vector<int> > dag;
public:
  // Make Dag of 'n' nodes, no edges for now.
  Dag( int n )
    : dag( vector< vector<int> >(n) ) {};
```

It's probably a good idea to have a function

```
const auto& neighbors( int i ) const { return dag.at(i); };
```

that, given a node, returns a list of the neighbors of that node.

Exercise 11.1. Finish the `Dag` class. In particular, add a method to generate example graphs:

- For testing the 'circular' graph is often useful: connect edges

$$0 \to 1 \to \cdots \to N-1 \to 0.$$

- It may also be a good idea to have a graph with random edges.

Write a method that displays the graph.

11.1.1.2 Node sets

The classic formulation of SSSP algorithms, such as the *Dijkstra shortest path algorithm* (see HPC book [6], section 9.1.3) uses sets of nodes that are gradually built up or depleted.

You could implement that as a vector:

```
vector< int > set_of_nodes(nnodes);
for ( int inode=0; inode<nnodes; inode++)
  // mark inode as distance unknown:
  set_of_nodes.at(inode) = inf;
```

where you use some convention, such as negative distance, to indicate that a node has been removed from the set.

However, C++ has an actual `set` container with methods for adding an element, finding it, and removing it; see section 24.3.2 (textbook). This makes for a more direct expression of our algorithms. In our case, we'd need a set of int/int or int/float pairs, depending on the graph algorithm. (It is also possible to use a `map`, using an int as lookup key, and int or float as values.)

For the unweighted graph we only need a set of finished nodes, and we insert node 0 as our starting point:

```
using node_info = std::pair<unsigned,unsigned>;
std::set< node_info > distances;
distances.insert( {0,0} );
```

For Dijkstra's algorithm we need both a set of finished nodes, and nodes that we are still working on. We again set the starting node, and we set the distance for all unprocessed nodes to infinity:

```
const unsigned inf = std::numeric_limits<unsigned>::max();
using node_info = std::pair<unsigned,unsigned>;
std::set< node_info > distances,to_be_done;

to_be_done.insert( {0,0} );
for (unsigned n=1; n<graph_size; n++)
  to_be_done.insert( {n,inf} );
```

(Why do we need that second set here, while it was not necessary for the unweighted graph case?)

Exercise 11.2. Write a code fragment that tests if a node is in the `distances` set.

- You can of course write a loop for this. In that case know that iterating over a set gives you the key/value pairs. Use *structured bindings*; section 24.5 (textbook).
- But it's better to use an 'algorithm', in the technical sense of 'algorithms built into the standard

library'. In this case, `find`.
- ... except that with `find` you have to search for an exact key/value pair, and here you want to search: 'is this node in the `distances` set with whatever value'. Use the `find_if` algorithm; section 24.3.2 (textbook).

11.1.2 Level set algorithm

We start with a simple algorithm: the SSSP algorithm in an unweighted graph; see HPC book [6], section 9.1.1 for details. Equivalently, we find level sets in the graph.

For unweighted graphs, the distance algorithm is fairly simple. Inductively:

- Assume that we have a set of nodes reachable in at most n steps,
- then their neighbors (that are not yet in this set) can be reached in $n+1$ steps.

The algorithm outline is

```
for (;;) {
  if (distances.size()==graph_size) break;
  /*
   * Loop over all nodes that are already done
   */
  for ( auto [node,level] : distances ) {
    /*
     * set neighbors of the node to have distance 'level + 1'
     */
    const auto& nbors = graph.neighbors(node);
    for ( auto n : nbors ) {
      /*
       * See if 'n' has a known distance,
       * if not, add to 'distances' with level+1
       */
/* ... */
      {
        cout << "node " << n << " level " << level+1 << '\n';
        distances.insert( {n,level+1} );
      }
```

Exercise 11.3. Finish the program that computes the SSSP algorithm and test it.

This code has an obvious inefficiency: for each level we iterate through all finished nodes, even if all their neighbors may already have been processed.

Exercise 11.4. Maintain a set of 'current level' nodes, and only investigate these to find the next level. Time the two variants on some large graphs.

11.1.3 Dijkstra's algorithm

In Dijkstra's algorithm we maintain both a set of nodes for which the shortest distance has been found, and one for which we are still determining the shortest distance. Note: a tentative shortest distance for a node may be updated several times, since there may be multiple paths to that node. The 'shortest' path in terms of weights may not be the shortest in number of edges traversed!

The main loop now looks something like:

```
for (;;) {
  if (to_be_done.size()==0) break;
  /*
   * Find the node with least distance
   */
  /* ... */
  cout << "min: " << nclose << " @ " << dclose << '\n';
  /*
   * Move that node to done,
   */
  to_be_done.erase(closest_node);
  distances.insert( *closest_node );
  /*
   * set neighbors of nclose to have that distance + 1
   */
  const auto& nbors = graph.neighbors(nclose);
  for ( auto n : nbors ) {
    // find 'n' in distances
  /* ... */
      {
        /*
         * if 'n' does not have known distance,
         * find where it occurs in 'to_be_done' and update
         */
  /* ... */
        to_be_done.erase( cfind );
        to_be_done.insert( {n,dclose+1} );
  /* ... */
}
```

(Note that we erase a record in the *to_be_done* set, and then re-insert the same key with a new value. We could have done a simple update if we had used a *map* instead of a *set*.)

The various places where you find nodes in the finished / unfinished sets are up to you to implement. You can use simple loops, or use *find_if* to find the elements matching the node numbers.

Exercise 11.5. Fill in the details of the above outline to realize Dijkstra's algorithm.

11.2 Linear algebra formulation

In this part of the project you will explore how much you make graph algorithms look like linear algebra.

11.2.1 Code preliminaries

11.2.1.1 Data structures

You need a matrix and a vector. The vector is easy:

```
class Vector {
private:
  vector<vectorvalue> values;
public:
  Vector( int n )
    : values( vector<vectorvalue>(n,infinite) ) {};
```

For the matrix, use initially a dense matrix:

```
class AdjacencyMatrix {
private:
  vector< vector<matrixvalue> > adjacency;
public:
  AdjacencyMatrix(int n)
    : adjacency( vector<vector<matrixvalue>>
                 ( n,vector<matrixvalue>(n,empty) ) ) {
  };
```

but later we will optimize that.

Remark 6 *In general it's not a good idea to store a matrix as a vector-of-vectors, but in this case we need to be able to return a matrix row, so it is convenient.*

11.2.1.2 Matrix vector multiply

Let's write a routine

```
Vector AdjacencyMatrix::leftmultiply( const Vector& left );
```

This is the simplest solution, but not necessarily the most efficient one, as it creates a new vector object for each matrix-vector multiply.

As explained in the theory background, graph algorithms can be formulated as matrix-vector multiplications with unusual add/multiply operations. Thus, the core of the multiplication routine could look like

```
for ( int row=0; row<n; row++ ) {
  for ( int col=0; col<n; col++) {
    result[col] = add( result[col], mult( left[row],adjacency[row][col] ) );
  }
}
```

11.2.2 Unweighted graphs

Exercise 11.6. Implement the `add` / `mult` routines to make the SSSP algorithm on unweighted graphs work.

11.2.3 Dijkstra's algorithm

As an example, consider the following adjacency matrix:

```
.  1  .  .  5
.  .  1  .  .
.  .  .  1  .
.  .  .  .  1
1  .  .  .  .
```

The shortest distance $0 \to 4$ is 4, but in the first step a larger distance of 5 is discovered. Your algorithm should show an output similar to this for the successive updates to the known shortest distances:

```
Input :  0 . . . .
step 0:  0 1 . . 5
step 1:  0 1 2 . 5
step 2:  0 1 2 3 5
step 3:  0 1 2 3 4
```

Exercise 11.7. Implement new versions of the `add` / `mult` routines to make the matrix-vector multiplication correspond to Dijkstra's algorithm for SSSP on weighted graphs.

11.2.4 Sparse matrices

The matrix data structure described above can be made more compact by storing only nonzero elements. Implement this.

11.2.5 Further explorations

How elegant can you make your code through operator overloading?

Can you code the all-pairs shortest path algorithm?

Can you extend the SSSP algorithm to also generate the actual paths?

11.3 Tests and reporting

You now have two completely different implementations of some graph algorithms. Generate some large matrices and time the algorithms.

Discuss your findings, paying attention to amount of work performed and amount of memory needed.

Chapter 12

Climate change

The climate has changed and it is always changing.
Raj Shah, White House Principal Deputy Press Secretary

The statement that climate always changes is far from a rigorous scientific claim. We can attach a meaning to it, if we interpret it as a statement about the statistical behavior of the climate, in this case as measured by average global temperature. In this project you will work with real temperature data, and do some simple analysis on it. (The inspiration for this project came from [11].)

Ideally, we would use data sets from various measuring stations around the world. Fortran is then a great language because of its array operations (see chapter 39 (textbook)): you can process all independent measurements in a single line. To keep things simple we will use a single data file here that contains data for each month in a time period 1880-2018. We will then use the individual months as 'pretend' independent measurements.

12.1 Reading the data

In the repository you find two text files

`GLB.Ts+dSST.txt` `GLB.Ts.txt`

that contain temperature deviations from the 1951–1980 average. Deviations are given for each month of each year 1880–2018. These data files and more can be found at https://data.giss.nasa.gov/gistemp/.

Exercise 12.1. Start by making a listing of the available years, and an array `monthly_deviation` of size $12 \times \mathrm{nyears}$, where nyears is the number of full years in the file. Use formats and array notation.

The text files contain lines that do not concern you. Do you filter them out in your program, or are you using a shell script? Hint: a judicious use of `grep` will make the Fortran code much easier.

12.2 Statistical hypothesis

We assume that mr Shah was really saying that climate has a 'stationary distribution', meaning that highs and lows have a probability distribution that is independent of time. This means that in n data points,

each point has a chance of $1/n$ to be a record high. Since over $n+1$ years each year has a chance of $1/(n+1)$, the $n+1$st year has a chance $1/(n+1)$ of being a record.

We conclude that, as a function of n, the chance of a record high (or low, but let's stick with highs) goes down as $1/n$, and that the gap between successive highs is approximately a linear function of the year[1].

This is something we can test.

Exercise 12.2. Make an array `previous_record` of the same shape as `monthly_deviation`. This array records (for each month, which, remember, we treat like independent measurements) whether that year was a record, or, if not, when the previous record occurred:

$$\mathrm{PrevRec}(m,y) = \begin{cases} y & \text{if } \mathrm{MonDev}(m,y) = \max_{m'}\big(\mathrm{MonDev}(m',y)\big) \\ y' & \text{if } \mathrm{MonDev}(m,y) < \mathrm{MonDev}(m,y') \\ & \text{and } \mathrm{MonDev}(m,y') = \max_{m''<m'}\big(\mathrm{MonDev}(m'',y)\big) \end{cases}$$

Again, use array notation. This is also a great place to use the `Where` clause.

Exercise 12.3. Now take each month, and find the gaps between records. This gives you two arrays: `gapyears` for the years where a gap between record highs starts, and `gapsizes` for the length of that gap.

This function, since it is applied individually to each month, uses no array notation.

The hypothesis is now that the gapsizes are a linear function of the year, for instance measured as distance from the starting year. Of course they are not exactly a linear function, but maybe we can fit a linear function through it by *linear regression*.

Exercise 12.4. Copy the code from `http://www.aip.de/groups/soe/local/numres/bookfpdf/f15-2.pdf` and adapt for our purposes: find the best fit for the slope and intercept for a linear function describing the gaps between records.

You'll find that the gaps are decidedly not linearly increasing. So is this negative result the end of the story, or can we do more?

Exercise 12.5. Can you turn this exercise into a test of global warming? Can you interpret the deviations as the sum of a yearly increase in temperature plus a stationary distribution, rather than a stationary distribution by itself?

1. Technically, we are dealing with a uniform distribution of temperatures, which makes the maxima and minima have a beta-distribution.

Chapter 13

Desk Calculator Interpreter

In this set of exercises you will write a 'desk calculator': a small interactive calculator that combines numerical and symbolic calculation.

These exercises mostly use the material of chapters 36 (textbook), 41 (textbook), 35 (textbook).

13.1 Named variables

We start out by working with 'named variables': the `namedvar` type associates a string with a variable:

```
type namedvar
   character(len=20) :: expression = ""
   integer :: value
end type namedvar
```

A named variable has a value, and a string field that is the expression that generated the variable. When you create the variable, the expression can be anything.

```
type(namedvar) :: x,y,z,a
x = namedvar("x",1 )
y = namedvar("yvar",2 )
```

Next we are going to do calculations with these type objects. For instance, adding two objects

- adds their values, and
- concatenates their `expression` fields, giving the expression corresponding to the sum value.

Your first assignment is to write `varadd` and `varmult` functions that get the following program working with the indicated output. This uses string manipulation from sections 35.3 (textbook) and 41.5 (textbook).

Exercise 13.1. The following main program should give the corresponding output:

Code:

```
print *,x
print *,y
z = varadd(x,y)
print *,z
a = varmult(x,z)
print *,a
```

Output
[structf] varhandling:

```
x                                1
yvar                             2
(x)+(yvar)                       3
(x)*((x)+(yvar))                 3
```

(To be clear: the two routines need to do both numeric and string 'addition' and 'multiplication'.)

You can base this off the file `namedvar.cpp` in the repository

13.2 First modularization

Let's organize the code so far by introducing modules; see chapter 37 (textbook).

Exercise 13.2. Create a module (suggested name: `VarHandling`) and move the `namedvar` type definition and the routines `varadd`, `varmult` into it.

Exercise 13.3. Also create a module (suggested name: `InputHandling`) that contains the routines `islower`, `isdigit` from the character exercises in chapter 35 (textbook). You will also need an `isop` routine to recognize arithmetic operations.

13.3 Event loop and stack

In our quest to write an interpreter, we will write an 'event loop': a loop that continually accepts single character inputs, and processes them. An input of `"0"` will mean termination of the process.

Exercise 13.4. Write a loop that accepts character input, and only prints out what kind of character was encountered: a lowercase character, a digit, or a character denoting an arithmetic operation `+-*/`.

Code:

```
do
   read *,input
   if (input .eq. '0') then
      exit
   else if ( isdigit(input) ) then
```

Output
[structf] interchar:

```
Inputs: 4 x 3 + 0
 4 is a digit
 x is a lowercase
 3 is a digit
 + is an operator
```

Use the `InputHandling` module introduced above.

13.3.1 Stack

Next, we are going to store values in `namedvar` types on a stack. A *stack* is a data structure where new elements go on the top, so we need to indicate with a *stack pointer* that top element. Equivalently, the stack pointer indicates how many elements there already are:

```
type(namedvar),dimension(10) :: stack
integer :: stackpointer=0
```

Since we are using modules, let's keep the stack out of the main program and put it in the appropriate module.

Exercise 13.5. Add the `stack` variable and the stack pointer to the `VarHandling` module.

Since Fortran uses 1-based indexing, a starting value of zero is correct. For C/C++ it would have been `-1`.

Next we will start implementing stack operations, such as putting `namedvar` objects on the stack.

13.3.2 Stack operations

We extend the above event loop, which was only recognizing the input characters, by actually incorporating actions. That is, we repeatedly

1. read a character from the input;
2. `0` causes the event loop to exit; otherwise:
3. if it is a digit, create a new `namedvar` entry on the top of the stack, with that value both numerically as the `value` field, and as string in the `expression` field.
   You may be tempted to write the following in the main program:

   ```
   if ( isdigit(input) ) then
      stackpointer = stackpointer + 1
      read( input,'( i1 )' ) stack(stackpointer)%value
      stack(stackpointer)%expression = trim(input)
   ```

   (You have already coded `isdigit` in exercise 35.1 (textbook).) but a cleaner design uses a function call to a method in the `VarHandling` module:

   ```
   else if ( isdigit(input) ) then
      call stack_push(input)
   ```

   ```
   subroutine stack_push(input)
     implicit none
     character,intent(in) :: input
   ```

   Note that the `stack_push` routine does not have the stack or stack pointer as arguments: since they are all in the same module, they are accessible as *global variable*.
   Finally,
4. if it is a letter indicating an operation $+$, $-$, $\times$, $/$,
   (a) take the two top entries from the stack, lowering the stack pointer;
   (b) apply that operation to the operands; and
   (c) push the result onto the stack.

The auxiliary function `stack_display` is a little tricky, so you get that here. This uses string formatting (section 41.3 (textbook)) and implied do loops (section 32.3 (textbook)): Also, note that the `stack` array and the `stackpointer` act like global variables.

```
subroutine stack_display()
  implicit none
  ! local variables
  integer :: istck
```

```
  if (stackpointer.eq.0) return
  print '( 10( a,a, a,i0,"; ") )', ( &
       " expr=",trim(stack(istck)%expression), &
       " val=",stack(istck)%value, &
       istck=1,stackpointer )

end subroutine stack_display
```

Let's add the various options to the event loop.

Exercise 13.6. Make your event loop accept digits, creating a new entry:

Code:

```
1 else if ( isdigit(input) ) then
2    call stack_push(input)
```

Output
[structf] internum:

```
Inputs: 4 5 6 0
 expr=4 val=4;
 expr=4 val=4;  expr=5 val=5;
 expr=4 val=4;  expr=5 val=5;  expr=6
```

Next we integrate the operations: if the `input` character corresponds to an arithmetic operator, we call `stack_op` with that character. That routine in turn calls the appropriate operation depending on what the character was.

Exercise 13.7. Add a clause to your event loop to handle characters that stand for arithmetic operations:

Code:

```
1 else if ( isop(input) ) then
2    call stack_op(input)
```

Output
[structf] internumop:

```
Inputs: 4 5 6 + + 0
 expr=4 val=4;
 expr=4 val=4;  expr=5 val=5;
 expr=4 val=4;  expr=5 val=5;  expr=6
 expr=4 val=4;  expr=(5)+(6) val=11;
 expr=(4)+((5)+(6)) val=15;
```

13.3.3 Item duplication

Finally, we may want to use a stack entry more than once, so we need the functionality of duplicating a stack entry.

For this we need to be able to refer to a stack entry, so we add a single character label field: the `namedvar` type now stores

1. a single character id,
2. an integer value, and
3. its generating expression as string.

```
type namedvar
   character :: id
   character(len=20) :: expression
   integer :: value
end type namedvar
```

Exercise 13.8. Add the `id` field to the `namedvar`, and make sure your program still compiles and runs.

The event loop is now extended with an extra step. If the input character is a lowercase letter, it is used as the `id` of a `namedvar` as follows.

- If there is already a stack entry with that `id`, it is duplicated on top of the stack;
- otherwise, the `id` of the stack top entry is set to this character.

Here is the relevant bit of the new `stack_print` function:

```
print '( 10( a,a1, a,a, a,i0,"; ") )', ( &
     "id:",stack(istck)%id, &
     " expr=",trim(stack(istck)%expression), &
     " val=",stack(istck)%value, &
     istck=1,top )
```

Exercise 13.9. Write the missing function and its clause in the event loop:

Code:

```
1 stacksearch = find_on_stack(stack,stackpointer,input)
2 if ( stacksearch>=1 ) then
3    stackpointer = stackpointer+1
4    stack(stackpointer) =  stack(stacksearch)
```

Output

```
[structf] stackfind:

Inputs: 1 x 2 y x y + z 0
id:. expr=1 val=1;
id:x expr=1 val=1;
id:x expr=1 val=1; id:. expr=2 val=2;
id:x expr=1 val=1; id:y expr=2 val=2;
id:x expr=1 val=1; id:y expr=2 val=2;
id:x expr=1 val=1; id:y expr=2 val=2;
    val=2;
id:x expr=1 val=1; id:y expr=2 val=2;
id:x expr=1 val=1; id:y expr=2 val=2;
```

(What is in the `else` part of this conditional?)

13.4 Modularizing

With the modules and the functions you have developed so far, you have a very clean main program:

```
do
   call stack_display()
   read *,input
   if (input .eq. '0') exit
   if ( isdigit(input) ) then
      call stack_push(input)
   else if ( isop(input) ) then
      call stack_op(input)
   else if ( islower(input) ) then
      call stack_name(input)
   end if
end do
```

You see that by moving the stack into the module, neither the stack variable nor the stack pointer are visible in the main program anymore.

But there is an important limitation to this design: there is exactly one stack, declared as a sort of global variable, accessible through a module.

Whether having global data is good practice is another matter. In this case it's defensible: in a calculator app there will be exactly one stack.

13.5 Object orientation

But maybe we do sometimes need more than one stack. Let's bundle up the stack array and the stack pointer in a new type:

```
type stackstruct
   type(namedvar),dimension(10) :: data
   integer :: top=0
 contains
   procedure,public :: display, find_id, name, op, push
end type stackstruct
```

Exercise 13.10. Change the event loop so that it calls methods of the `stackstruct` type, rather than functions that take the stack as input.

For instance, the `push` function is called as:

```
if ( isdigit(input) ) then
   call thestack%push(input)
```

13.5.1 Operator overloading

The `varadd` and similar arithmetic routines use a function call for what we would like to write as an arithmetic operation.

Exercise 13.11. Use *operator overloading* in the `varop` function:

```
if (op=="+") then
   varop = op1 + op2
```

et cetera.

PART III

APPENDIX

Chapter 14

Style guide for project submissions

The purpose of computing is insight, not numbers. (Richard Hamming)

Your project writeup is equally important as the code. Here are some common-sense guidelines for a good writeup. However, not all parts may apply to your project. Use your good judgment.

14.1 General approach

As a general rule, consider programming as an experimental science, and your writeup as a report on some tests you have done: explain the problem you're addressing, your strategy, your results.

Turn in a writeup in pdf form (Word and text documents are not acceptable) that was generated from a text processing program such (preferably) LaTeX. For a tutorial, see Tutorials book [5], section-15.

14.2 Style

Your report should obey the rules of proper English.

- Observing correct spelling and grammar goes without saying.
- Use full sentences.
- Try to avoid verbiage that is disparaging or otherwise inadvisable. The *Google developer documentation style guide* [7] is a great resource.

14.3 Structure of your writeup

Consider this project writeup an opportunity to practice writing a scientific article.

Start with the obvious stuff.

- Your writeup should have a title. Not 'Project', but something like 'Simulation of Chronosynclastic Enfundibula'.
- Author and contact information. This differs per case. Here it is: your name, EID, TACC username, and email.
- Introductory section that is extremely high level: what is the problem, what did you do, what did you find.

- Conclusion: what do your findings mean, what are limitations, opportunities for future extensions.
- Bibliography.

14.3.1 Introduction

The reader of your document need not be familiar with the project description, or even the problem it addresses. Indicate what the problem is, give theoretical background if appropriate, possibly sketch a historic background, and describe in global terms how you set out to solve the problem, as well as a brief statement of your findings.

14.3.2 Detailed presentation

See section 14.5 below.

14.3.3 Discussion and summary

Separate the detailed presentation from a discussion at the end: you need to have a short final section that summarizes your work and findings. You can also discuss possible extensions of your work to cases not covered.

14.4 Experiments

You should not expect your program to run once and give you a final answer to your research question.

Ask yourself: what parameters can be varied, and then vary them! This allows you to generate graphs or multi-dimensional plots.

If you vary a parameter, think about what granularity you use. Do ten data points suffice, or do you get insight from using $10,000$?

Above all: computers are very fast, they do a billion operations per second. So don't be shy in using long program runs. Your program is not a calculator where a press on the button immediately gives the answer: you should expect program runs to take seconds, maybe minutes.

14.5 Detailed presentation of your work

The detailed presentation of your work is as combination of code fragments, tables, graphs, and a description of these.

14.5.1 Presentation of numerical results

You can present results as graphs/diagrams or tables. The choice depends on factors such as how many data points you have, and whether there is an obvious relation to be seen in a graph.

Graphs can be generated any number of ways. Kudos if you can figure out the LaTeX `tikz` package, but Matlab or Excel are acceptable too. No screenshots though.

Number your graphs/tables and refer to the numbering in the text. Give the graph a clear label and label the axes.

14.5.2 Code

Your report should describe in a global manner the algorithms you developed, and you should include relevant code snippets. If you want to include full listings, relegate that to an appendix: code snippets in the text should only be used to illustrate especially salient points.

Do not use screen shots of your code: at the very least use a monospace font such as the `verbatim` environment, but using the `listings` package (used in this book) is very much recommended.

14.6 General index

Chapter 15

Bibliography

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings Supercomputing '90*, pages 2–11. IEEE Computer Society Press, Los Alamitos, California, 1990.

[2] Roy M. Anderson and Robert M. May. Population biology of infectious diseases: Part I. *Nature*, 280:361–367, 1979. doi:10.1038/280361a0.

[3] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209 – 219, 2006.

[4] https://github.com/jarro2783/cxxopts.

[5] Victor Eijkhout. HPC carpentry. https://theartofhpc.com/carpentry.html.

[6] Victor Eijkhout. The science of computing. http://theartofhpc.com/istc.html.

[7] Google. Google developer documentation style guide. https://developers.google.com/style/.

[8] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):1–25, 2008.

[9] https://mathworld.wolfram.com/Kermack-McKendrickModel.html.

[10] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[11] Juan M. Restrepo, , and Michael E. Mann. This is how c̈limate is always changing.̈ *APS Physics, GPS newletter*, February 2018.

[12] Lin S. and Kernighan B. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.