Data Structures Notes

Taught by John Nevard

Daniel Kim, Sameer Pai

December 9, 2019

These are a collection of notes (typed live) of all classes of Data Structures taught by John Nevard at the Bergen County Academies.

Contents

§1 Review of C++

§1.1 Reference

In Java, everything is a **reference**, while C++ is mainly **pass-by-value**. For example, consider the following snippet of Java code:

```
Array <Int> x = new Array<Int>(17);
Array <Int> y = x;
```

Here, `x` and `y` refer to the same block of memory (that is, the array of integers of length 17). Meanwhile, in C++ , if we have:

```
vector<int> x(17);
vector<int> y = x;
```

Then, `x` and `y` refer to separate arrays of integers of length 17 respectively.

§1.2 Operators

In C++ , we have the following operators, in order of precedence:

1. `::` (the scope resolution operator), $::_1$

   For example, this operator allows us to access static variables of classes:

   ```
   class X {
      static int a;
   };
   int y = x::a
   ```

   Henceforth, we use the subscript $_1$ to refer to the same operator in a **unary** context, meaning that it requires only one argument.

2. `[]`, `()`

3. $*_1$, $\&_1$, `!`, `~`, $+_1$, $-_1$, `new`, `delete`, `delete[]` (which is used for arrays of objects), `++`, `--`

4. `+`,`-`

5. `==`, `!=`

6. `>`, `>=`, `<`, `<=`

7. `&`

8. `^`

9. `|`

10. `&&`

11. `||`

12. `?` `:` (**ternary choice**)

    For example, consider the following code:

```
    double y;
    if (x > 0) {
        y = sqrt(x);
    } else {
        y = sqrt(-x);
    }
```

We can significantly condense this using a ternary opertor:

```
    double y = x > 0 ? sqrt(x) : sqrt(-x);
```

13. `=`, `+=`, `-=`, `*=`, `/=`, `%=`

    For example,

```
    a += b
    a = a + b
```

    These two lines do the exact same thing.

§1.3 Short Circuit

Consider the following condition:

```
if (i >= 0 && a[i] < 0) {
```

If the computer evaluates `i >= 0` to be false, then the whole condition will be false no matter what (because of logical AND). Thus, the computer saves time by just ignoring the rest of the statement (namely `a[i] < 0`) and continues on with the program. In this fashion, we can avoid accessing an array out of bounds.

§1.4 Bit Operators

An `int` is usually 4 bytes. Its formal keyword is `int_32`. A `long` is usually 8 bytes. Its formal keyword is `int_64`. An `unsigned int` also has 8 bytes, and its formal keyword is `uint_64`.

§1.5 Trivial I/O

To provide us with basic input and output, we first include the line

```
#include <iostream>
```

If you ONLY include this line, then you will have to write

```
std::cout << "Hello World!\n";
```

to print `Hello World!`. Otherwise, if we start with:

```
#include <iostream>
using namespace std;
```

Then, we can simply write:

```
cout << "Hello World!\n";
```

to do the same thing. However, it is recommended not to use `using namespace std`. Instead, one should write:

```
#include <iostream>
using std::cout;
```

using the scope resolution operator. This enables us to do the same as before.

Here is an example of how `cout` allows us to print concatenations of variables of various data types:

```
int x = 17;
double y = 3.14;
string s = "Huh?";
cout << x << ' ' << y << ' ' << s << '\n';
```

This prints: `17 3.14 Huh?`.

Furthermore, we can conveniently read input using `cin`. The following code takes in two integers and a string provided by the user in that order:

```
int x, y;
string s;
cin >> x >> y >> s;
```

§1.6 "Hello World"

Here is the classic "Hello World" program for C++ :

```
#include <iostream>
using std::cout;
int main() {
    cout << "Hello World\n";
    return 0;
}
```

§1.7 Preprocessor Directive

Consider the following block of code:

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
    cout << "Enter x & y: ";
    int x;
    double y;
    cin >> x >> y;
    cout << "x + y = " << x+y << '\n'; // Use endl if you want to flush the output
    return 0;
}
```

There's also *spooky* I/O:

```
#include <cstdio> // C I/O
// "stdio.h" is the same as "cstdio"
int main() {
    // More efficient than cout
    printf("HI!");
    int x;
    scanf("%d", &x);
    return 0;
}
```

Now, the `#` in `#include <iostream>` is called a **preprocessor directive**. There are many different possible preprocessor instructions that can be used after the `#`, but here the "include" statement tells the compiler to add the functions and classes another file to the code. Namely, `iostream` is called a **header file**.

We can also include custom header files (not in the standard library) by using double quotes, i.e. `#include "my_cool_stuff.h"`.

To prevent redefinitions of variables and functions, we use something called a **include guard**. How this works is in each header file we use a "preprocessor if statement" to ensure the file has not been included in the past.

```
#ifndef __my_cool_stuff__ // Only #include once
#define __my_cool_stuff__
// code here
#endif
```

§1.8 Variables

Another important keyword is `const`. This keyword means that the variable we declare cannot change, ever. If we were to write:

```
const double PI = 3.14159;
PI = 3; // Oops!
```

This would fail, because we already said that PI must be 3.14159. A good practice in programming is to use `const` whenever possible.

A difference between `#define` and `const` is memory usage: the preprocessor will generally use less memory, but the peace of mind granted by `const` more than makes up for this.

§1.9 Function Prototypes

Again, we point to another fundamental difference between Java and C++ : objects. In C++ , we can declare a function before explicitly writing the code for it. The way we do it is a **function prototype**.

```
int f (double x); // Prototypes

// Rest of code

y = f(3.7);
```

Even though the compiler does not know exactly what `f` does, it knows what its input and return types are from the prototype, so it can still use `f`, even if it is defined in another file.

In most cases, you can find a way to order your functions so that prototypes are not needed. However, if you have a "loop," then you need to write some prototypes. Consider the example:

```
int g(); // Necessary
int f(); // Optional

int f() {
    g();
}

int g() {
    f();
}
```

Here, both functions call each other, so in order to use a function before it is defined, we need to utilize at least one prototype.

§1.10 Swap 2 Values

There are many ways to swap two values in C++ . One is the `swap` function included in `<algorithm>`:

```
#include <algorithm>
using std::swap;

// Rest of code

swap(x,y);
```

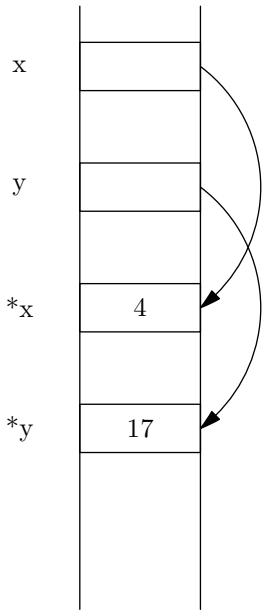
But how was `swap` even implemented? Let's first try (but fail to do so):

```
// wrong code
void swap (int x, int y) { // For now, assume we are swapping int variables
    int t = x;
    x = y;
    y = t;
}
```

The problem is that `t`, `x`, `y` are **local variables**. C++ is **pass-by-value**, meaning that any variables passed in to a function have their values copied to another location in memory before running the function. So, any changes to `x` and `y` are not actual changes to the original variables.

The problem with memory might encourage us to use pointers instead (the unary asterisk declares that `x` is a pointer to an integer in memory):

```
void swap (int *x, int *y) {
    int t = *x; // dereference the pointer
    *x = *y; // Replace contents of address x by contents of address y
    *y = t; //
}

int u = 3, v = 17;
swap(&u, &v); // Unary ampersand takes the memory address of the variable
```



Although this works, it's annoying to write `&x` and `&y` instead of plain `x` and `y`. Thus, our last attempt uses references.

```
void swap(int& x, int& y) { // int& is a reference to int (aliases)
    int t = x;
    x = y;
    y = t;
}
int u = 3, v = 17;
swap(u,v); // This will work!
```

§1.11 Classes

Now, we discuss the `class` in C++ . Consider the following example of code:

```
class BUC {
    // Code here
};

void f(BUC b1, BUC b2); // prototype
BUC buc1, buc2;

// Wasteful code
f(buc1, buc2); // Copies buc1 to b1, and buc2 to b2
```

However, we can use references to avoid copying these values:

```
class BUC {
    // Code here
};

void f(BUC& b1, BUC& b2); // prototype
BUC buc1, buc2;

f(buc1, buc2);
```

However, let's say that the operations `+`, `-` were defined for our class `BUC`. Then,

```
f(b1+b2, b1-b2);
```

would not compile. First, the compiler would look up the code for `+`, `-` for the class `BUC`, and then find the values for `b1+b2` and `b1-b2`, and store them in respective places in memory. But if we take the references of these values, the compiler would not know where to look for in memory.

In fact, `b1+b2` and `b1-b2` are not **Lvalues**, meaning that they are not things that can be assigned to. Some examples of Lvalues include:

```
x = ...
*x = ...
x[1] = ...
*x[i+j] = ...
i + j = k \\ Not valid
```

In essence, an Lvalue is a thing that can appear on the left side of an assignment `=`.

So how do we fix a function to where we can pass in objects that are *not* Lvalues? The way we do this is the use of the `const` keyword:

```
void f(const BUC& b1, const BUC& b2) {
    cout << b1;
    cout << b2;
}
```

The `const` keyword tells the compiler that we do not intend to change the two BUCs. This means we can pass in non-Lvalues without worrying about a compiler error. The restriction is that we cannot change the values in the function.

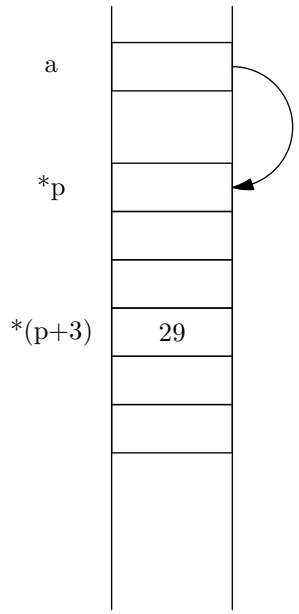
§1.12 Arrays

How do we declare an array in C++ ? We can use the following code:

```
int a[17];
int *p = a;
```

In this code, when we declare the array `a`, it is necessary that the *size specifier* inside the brackets is a constant expression.

When we declare a pointer and assign it to the array `a` of size 17, the pointer will point to the first element of the array. However, if we then take `p+3`, we are adding 3 times the size of `int` to the pointer, so `p+3` will now point to the fourth element in `a`. In terms of memory addresses, if `p` = 1000, then `p+3` = 1012.



If we wanted to assign the fourth element to 29, then the following lines would all do the same thing:

```
int a[17];
int *p = a;
*(p+3) = 29;
p[3] = 29;
a[3] = 29;
3[a] = 29
```

Furthermore, we note that `f(int a[])` and `f(int *a)` are equivalent. However, passing this argument alone leaves us with a major flaw:

```
f(int a[]) {
    a[5] = 53; // No clue if enough space!
}
```

We don't know if the array `a` even has a fifth element available. Thus, we usually specialize the number of elements in `a`, as such:

```
f(int a[], int na) {
    // Rest of code
}
```

Another way to define an array is using the **new** keyword. For example:

```
int *a = new int[17];


delete[] a; // no automatic garbage collection
```

The `new` keyword has the advantage that your allocated memory will never go out of scope: it will remain there until you `delete` it. Also of note is that whenever you use **heap allocation**, the memory you allocate has a *header* which is not part of the data.

Using `new` comes with the caveat of having to garbage collect your own variables as well. Unlike Java, C++ does not automatically garbage collect (the onus is on you).

```
class Dog {
    // Insert dog code
};

Dog *sirius = new Dog();
Dog *pack = new Dog[17];
// Do stuff
delete sirius;
delete[] pack;
```

It is good practice to never use `new` outside of classes.

§1.13 `vector`

However, one data structure usually works much better than just a normal array: the `vector`. To be able to start using it, make sure you include the line:

```
#include <vector>
using std::vector;

int main() {
    vector<int> a(17); // Don't do a[17]! That would make an array of 17 vectors!
    vector<int> a(8, 5) // 8 elements initialized to 5.
    a[3] = 42;
    a[17] = 100; // error!
}
```

When the vector goes out of scope, its destructor is called automatically, and each of the allocated members of the array are deleted automatically. So, if we use classes correctly, we do not have to worry about memory leaks.

To make our lives easier, we can make a shortcut for `vector<int>`. We can also declare multi-dimensional vectors. The following code demonstrates these:

```
using VI = vector<int> // or, typedef vector<int> VI;
using VD = vector<double>;

vector<VD> m(100, VD(100)); // 100 x 100 matrix of doubles, all initiated to 0

m[3][17] = 3.14159;

double determinant(const vector<VD>& m) {
    double det;
    vector<VD> m_copy = m;
    // stuff that changes m_copy, not m
    return det;
}
```

Alternatively, instead of declaring `m_copy`, we could write

```
double determinant(vector<VD> m) {
    double det;
```

```
    // stuff that changes m
    return det;
}
```

but that would be bad practice since we are modifying the argument that is passed in. Certainly, if we write:

```
double determinant(const vector<VD> m) {
    double det;
    vector<VD> m_copy = m;
    // stuff that changes m_copy, not m
    return det;
}
```

This code is wasteful, since not passing by reference already makes a copy of `m`.

§1.13.1 Useful member methods in `vector`

```
using VD = vector<double>;
int main() {
    VD a(17);
    cout << a.size(); // outputs 17
    a.clear(); // a is now empty
    cout << a.size(); // 0
    a.resize(1000); // changes a's size to 1000

    VD b;
    bool done = false;
    while(!done) {
        double x;
        cin >> x;
        done = (x == PI);
        b.push_back(x); // adds x to the end of b
    }

}
```

For `a.resize(1000);`, this will leave the first thousand (or less) elements in vector `a` alone, while the data beyond that will be destroyed.

But how do `push_back` and `resize` work? In each `vector` object, there is an underlying array of some size. If there is still room in the underlying array, we simply add the element to the array.

However, if not, we *reallocate* all the elements into an array of twice the size, then add the necessary elements.

If we calculate (which we might do later), we will find that this only takes twice as much time as a static array, which is as good as we can get.

§1.14 Classes in Java and C++

As Java is an object-oriented language, it supports inheritance:

```
// Inheritance
class Dog : public Mammal {
}
```

Suppose we write the following class in C++ (note that this is a *header file*):

```
// File: Dog.h
using std::string;

class Dog {
```

```
    public:
    // Constructor
    Dog(const string& name,
        const string& breed = "mutt", // default arg
        const int& age = 0); // default arg

    // Destructor
    ~Dog(); // no arguments

    private:
    string name;
    string breed;
    int age;
    const int id;
};
```

The differences from Java are that we need a destructor (which has no arguments) that de-allocates all memory. We also do not implement the methods in the class: we are allowed to do it in another file (Here, we will call it Dog.cpp). In general, header files only declare methods, and they are implemented in a regular C++ file.

Now we demonstrate the implementation of class `Dog`:

```
// File: Dog.cpp

// Definition
Dog::Dog(const string& name,
         const string& breed,
         int age,
         int id) // cannot put default args in definition
: name(name), // first "name" refers to "name" in private attributes, second "name"
    refers to the string passed in as arg
  breed(breed),
  age(age),
  id(17) // ctor initializers
 {
    // Rest of code
 }

Dog:: ~Dog() { } // Always have destructor even if it's empty
```

§1.15 Some Diagnostic Review

Horner's method for evaluating polynomials with n multiplications:

```
double eval(double x, double a[], int n) {
    double y = a[--n];
    while (--n >= 0)
        y = a[n] + x * y;
    return y;
}
```

Merge two sorted arrays:

```
void merge(double c[], double a[], double b[], int na, int nb) {
    int i = 0, j = 0, k = 0;
    while (i < na && j < nb) {
        c[k++] = (a[i] <= b[j] ? a[i++] : b[j++]);
    }
    while(i < na)
        c[k++] = a[i++];
    while(j < nb)
        c[k++] = b[j++];
```

```
}
```

The functions below are ordered by rate of growth:

$$2^{2^{n+1}}, 2^{2^n}, n!, e^n, 2^n, n^{\log\log n}, n^3, n\log n, \sqrt{2}^{\log n} = \sqrt{n}, \sqrt{\log n}, \log\log n, n^{\frac{1}{\log n}} = 2$$

§1.16 Operator Overloading

Consider making a class `R3`, consisting of three-dimensional points in $\mathbb{R}^3$. We want to be able to use the points in convenient ways, like this:

```
R3 p (1,2,3), q(2, 5, 6);
cout << 2*p + 3*q; // Want to be able to do this with impunity
// Desired console output: (8, 19, 24)
```

So can we "modify" addition and `cout` to make this work? Consider the following class implementation:

```
class R3 {
public:
    R3(double x = 0, double y = 0, double z = 0): x(x), y(y), z(z) {} // Stroustrup says no!

    // Add two R3 class members together
    R3 operator+(const R3& p) const { // Compiler chooses this when invoked p is const
        return R3(x+y.x, y+p.y, z+p.z);
    }
    R3 operator+(const R3& p) { // Otherwise, this one
    }

    // Multiply an R3 class member by scalar
    friend R3 operator*(double t, const R3& p) // Not a member of the class; the double t is the "family friend"
    {
        return R3(t*p.x, t*p.y, t*p.z);
    }

    // Allow console to print an R3 class member in some format
    friend ostream& operator<<(ostream& os, const R3& p);


    ~R3();

private:
    double x, y, z;
};

ostream& operator<<(ostream& os, const R3& p) {
    return os << "(" << p.x << " " << p.y << " " << p.z << ")";
}

R3 operator*(double t, const R3& p) {
    return R3(t*p.x, t*p.y, t*p.z);
}
```

Some new syntax used in this code: when we put `const` after a member function, it means that the object that the function is called on never changes.

Before we continue with operator overloading, here are some functions that should always be in any class:

```
class X {
    X(); // default constructor
```

```
    X(double x = 0, double y = 0, double z = 0); // still counts as default constructor
        since all parameters are optional
    // Cannot have both constructors; choose one

    X(const X& x); // copy constructor

    X& operator=(const X& x); // Assignment overload
};

vector<X> a(17); // needs default constructor
f(X x); // calls copy constructor
X x,y,z;
y = x = z; // Assignment parses right to left: y = (x = z)
// Then we need to pass back a reference to x to assign something to y
```

When an assignment operator is not explicitly defined, the compiler tries to make a "basic" one for you. However, this can have unintended effects. As we discussed earlier, a vector actually contains a pointer to an array, and not the array itself. So, if we were to use the "basic" assignment for vectors, it would just copy the pointer over, and not the actual data. Therefore, this would make a *shallow copy*, where changes in one object cause changes in the other.

§1.17 Assignment Operators

Now, we discuss how to define assignment operators for custom classes:

```
class R3 {
    // rest of code for R3 class

    // Overloaded versions must be members
    R3& operator+= (const R3& p) { // a += b is an Lvalue
        x += p.x;
        y += p.y;
        z += p.z;
        return *this; // Reference
    }

    R3 operator+ (const R3& p) const {
        return R3(x+p.x, y+p.y, z+p.z);
    }
    // Either we can define + inside R3, or...
};

// OR: outside of R3, we can define binary operator for +
R3 operator+ (const R3& p, const R3& q) {
    R3 r = p; // copy constructor
    r += q; // calls assignment operator
    return r; // calls copy constructor ("move constructor")
}
```

Let's do another elaborate example:

```
// A complex number class

class Complex {
public:
    Complex(double x = 0.0, double y = 0.0)
    : x(x), y(y) {}

    Complex(const Complex& z): x(z.x), y(z.y) {}

    Complex& operator=(const Complex& z) {
```

```
        x = z.x;
        y = z.y;
        return *this;
    }

    Complex& operator/=(const Complex& z) {
        double norm = z.x*z.x + z.y*z.y;
        double nx = (x * z.x + y * z.y)/norm;
        double ny = (y * z.x - x * z.y)/norm;
        x = nx;
        y = ny;
        return *this;
    }

    // similar for +=, *=, -=

    Complex operator-() const {
        return Complex(-x, -y);
    }
private:
    double x, y;
};

Complex operator+ (const Complex& z, const Complex& w) {
    Complex sum = z;
    sum += w;
    return sum;
}
```

A 1-parameter constructor is a **conversion operator**. For example, if we call `Complex z1(17);`, we are converting a double, 17, to a complex number. This renders something like the following legal:

```
Complex w;
w = 3;
```

However, this feature can be disabled. For example,

```
vector<int> a(17), b;
b = 17; // Illegal
```

the compiler might assume that from the 1-parameter constructor `vector<int> a(17)`, we are able to convert from a double to a vector. But in the `vector` class,

```
class vector {
    // Rest of code
    explicit vector(size_t n_elts, T t = 0) { ... }
}
```

we explicitly define the constructor for `vector`. This disallows the conversion constructor.

If we wanted to override the ostream `<<` operator for this class, we would write

```
ostream& operator<< (ostream& os, const Complex& z) {
    return os << z.re() << " + " << z.im() << "i";
}
```

although this implementation would not account for complex numbers like $2 + 0i$ or $0 + 1i$.

We can also override the input for this class (i.e. we can directly read a Complex number from the user):

```
istream& operator>>(istream& in, Complex& z) {

}
```

§1.18 Templates

Templates allow us to generalize the type of arguments passed into a function.

```
template<typename T>
void swap(T & x, T & y) {
    T t = x; // Needs copy constructor
    x = y; // Needs assignment operator
    y = t; // Needs assignment operator
}

// Working code
long x = 17, y = 39;
int z = 3, w = 5;
swap(x,y);
swap(z,w);

swap(z,x); // Error because different data type
```

In fact, the class `vector` uses a template:

```
template<typename T>
class vector {
private:
    size_t size;
    size_t cap;
    T* ptr;
}

vector<int> a(17);
vector<double> b(23);
vector<Mod> c(117);
vector<Dog> d;
Dog fido(...);
d.push_back(fido);
```

```
#include <utility>

template<typename F, typename S>
class pair {
public:
    F first;
    S second;
    pair(const F &first, const S &second): first(first), second(second) {}
};

using PSD = std::pair<string, double>;
PSD x("john", 17);
```

```
template<class T>
class Sortable {
public:
    // other code
    bool operator<(const Sortable& x) const;
}
```

Templates encourage **code reuse**, which is the principle of reusing code as much as possible, a good practice in programming.

§1.19 Pointers to Objects

```cpp
class X {
public:
    void f();
    int g();
private:

}

X *p = new X();
// Instead of writing (*p).f();
p->f();
p->a;
```

§1.20 Functors

A class with a function call can be overloaded. This class would then be called a **functor**. Consider, for example, this Polynomial class:

```cpp
#include <iostream>
#include <vector>
using VD = vector<double>;
using std::cout;

class Poly {
public:
    Poly(const VD& coeffs); // c[0]+c[1]*x+...+c[n-1]x^{n-1}
    double operator()(double x) const; // use Horner's method
private:
    VD c;
};

template<typename F, typename D>
double newton_solve(double x0, const F& f, const D& df) {
    for(int i = 0; i < MAX; ++i) {
        double dx = f(x0)/df(x0);
        if (fabs(df(x0)) < EPS * fabs(df(x0))) {throw new BAD_NEWTON();}
        if(fabs(dx) < EPS * fabs(x0)) return x0;
        x0 -= dx;
    }
    cerr << "Max iterations exceeded!\n";
    throw new BAD_NEWTON();
    return 0; // rage against the machine
}

int main() {
    Poly q(VD{2, 3, 5}); // q = 2 + 3x + 5x^2
    cout << q(4) << '\n';
}
```

```cpp
#include <algorithm>

using VD = vector<double>;
using FID = double(*)(int); // Pointer to a function taking an int arg returning a
    double
using FDDB = bool(*)(double, double);

template<typename I, typename Cmp>
void sort(I beg, I end, Cmp cmp = ...) { // where cmp is arbitrary compare functor
//code
    if(cmp(*p, *q) < 0) {
        // Code here
    }
}
```

```
14
15
16 FDDB p = my_cmp; // passing a function without parentheses makes it a function pointer;
17 VD a(17);
18 sort(a.begin(), a.end(), p);
```

§1.21 Iterators

An **iterator** is an object with certain functions overloaded:

1. The unary `*` operator
2. The unary `++` operator
3. The unary `--` operator
4. The unary `[]` operator (subscripting)

An iterator with the first two operators a **forward iterator**. An iterator with the first three is called a **bidirectional iterator**. If the iterator also supports subscripting, it is a **random-access iterator**.

Iterators can also be classified into **write-only** and **read-only**, depending on whether the `*` operator returns an Lvalue or not.

Note that a built-in pointer is an iterator! So, we can do the following:

```
1 int a[] = {3,1,4,1,5,9};
2 sort(a,a+6); // also sorts entire array
```

A keyword useful when dealing with iterators is `auto`. It tells the compiler to figure out what type to use.

```
1 template<class V>
2 void f(const V& v) {
3     // V::const_iterator it = v.begin();
4     auto it = v.begin();
5 }
```

§2 Structures

§2.1 Linked Lists

§2.1.1 Definition

Linked lists are put together with each element having not just content to store but also a `next` pointer which points to the next element in the list.

```
template <typename T>
struct Link {
    explicit Link (const T& info, Link *next=nullptr) : info(info), next(next) {}
    ~Link() { delete next; } // deletes the whole list
    // If you just want to delete one element, set next=nullptr first and then delete
        next

    T info;
    Link *next;
};

Link<string> *head = new Link<string>("hi!");
vector<string> vs {"one","two","three"};
for (int i = 0; i < vs.size(); ++i) {
    head = new Link<string>(vs[i], head);
}
// "three" --> "two" --> "one" --> "hi"
```

Linked lists often are not "allowed" to be empty. Instead, we use a special element called a **sentinel**. A sentinel is an element of the linked list that signifies that we have reached the end of the list.

§2.1.2 Circular Linked Lists

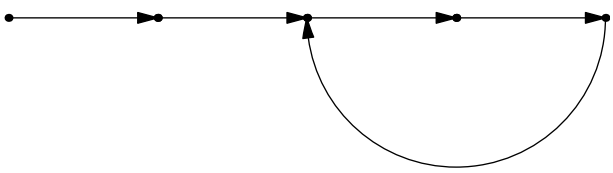
A **circular linked list** is a linked list whose end element's next pointer points to the "head," creating a "circular" chain of elements. Thus, it can be implemented with our without a "head."

§2.1.3 Doubly-Linked Lists

Doubly-Linked Lists are just like regular linked lists, except there is also a `prev` pointer which points to the previous element in the chain.

§2.1.4 Floyd's Algorithm

Floyd's algorithm is an efficient way to detect loops in a linked list. It is not hard to see that the only possible way a linked list can loop is as follows:



We call the **tail length** the number of links that are not part of the loop, and the **cycle length** the number of links that are part of the loop. In the above graph, for example, the tail length is 2 and the cycle length is 3. How can we determine:

1. Whether there is a loop?

2. If it exists, what is the cycle length?

3. What is the tail length?

We use **Floyd's Algorithm** to find these three details: Start with two pointers, one "slow" (call it `s`) and one "fast" (call it `f`). First set `s` and `f` at the starting node.

1. Increment `s` one link at a time, and `f` two links at a time, until `f == nullptr` (then there is no loop) or `f == s` (then there is a loop).

2. Determine cycle length: Advance `s` until `s == f`.

3. Determine length of tail: Move `s` and `f` to the beginning. Move `f` the cycle number of times. Then move `s` and `f` one link at a time and count until `s == f`.

What is the runtime of this algorithm? We reach the end of the tail after moving `s` t times and `f` $2t$ times. Then, since initially the distance between s and f is t links, and each move brings them one closer, the number of moves needed to bring the two together is $(-t) \pmod{c} \leq c$ turns, which is actually $3c$ moves. So the total number of moves for step 1 is $3(c+t)$ at most. Therefore, it is $O(t+c)$, or linear in the total number of links. Clearly, step 2 takes c moves. Finally, step 3 has $c+2t$ moves in total: c to set up f and $2t$ to reach the end of the tail. In total, the entire algorithm takes less than $5n$ moves.

§2.1.5 Brent's Algorithm

Let $\{a_n\}_{n=0}^{\infty}$ be strictly increasing positive integers. Then:
Start with two pointers, s and f. For all integers i, starting at 0 and increasing:

1. Increment f a_i times, stopping if f becomes null or `s == f`.

2. Set `s = f`.

In practice, we set $a_n = 2^n$.
This algorithm is generally faster than Floyd's algorithm, even though we will not prove anything about its runtime.

§2.1.6 The Josephus Problem

The following theoretical problem is based on a tale of Joseph from Greek mythology. Consider a circle of n people in a circle, labeled $0, \ldots, n-1$. Start from person 0, and go around the circle in order, going to the next kth person and "kill" that person. For example, if we have $n = 10$ and $k = 3$, we would kill person 3 first. After person 3 is killed, we move along the circle another 3 times and kill person 6. Similarly, person 9 is killed as well. But at this point, the only people surviving are

$$0, 1, 2, 4, 5, 7, 8.$$

After killing person 9, we count 3 people *out of the surviving ones* and kill person 2 next. Then, continuing until all people in the circle are killed, the **Josephus sequence** for $n = 10$ and $k = 3$ would be

$$3, 6, 9, 2, 7, 1, 8, 5, 0, 4,$$

the order in which the people are killed (i.e. person 4 would be the last to get killed).

The idea to solve for the Josephus sequence is to use a linked list for the circle of people, and eliminate one person at a time.

§2.2 Stacks

Before talking about data structures, we must start with a definition:

Definition 2.1. An **Abstract Data Type (ADT)** is a structure whose behavior is specified, but not the implementation.

```
#include <stack>

stack<int> s;
s.push(3);
s.push(17);
while (!s.empty()) {
    int t = s.top(); s.pop(); // rage against the machine
    cout << t << '\n';
}
```

From this code we see what we want a stack to be able to do:

1. Create an empty instance
2. Test if empty
3. Add an element to stack
4. Remove most recently added item

One possible implementation of the stack uses the linked list. We can let the stack point to the head of the list.

```
class Stack {
public:
    void push(int x) {
        stack = new Link(x, stack);
    }
    int top() const {
        if (!stack)
            throw new EmptyStackException();
        return stack->info;
    }
    void pop() {
        if (!stack) {
            throw new EmptyStackException();
        }
        Link* n = stack->next;
        delete stack;
        stack = n;
    }
private:
    Link* stack;
}
```

However, storing all of those next pointers is inefficient. A better implementation is as an array: We keep an array of integers, with a `stack` pointer pointing to the top element. To add an element we do: `*stack++ = x`. To pop an element we do: `return *--stack`.

We maintain a stack pointer which always points to the end of the stack, which is just many stack frames put together. A **stack frame** is a block of memory which keeps track of, in order:

- Return address

- Return value
- Parameters
- Pointer to next stack frame

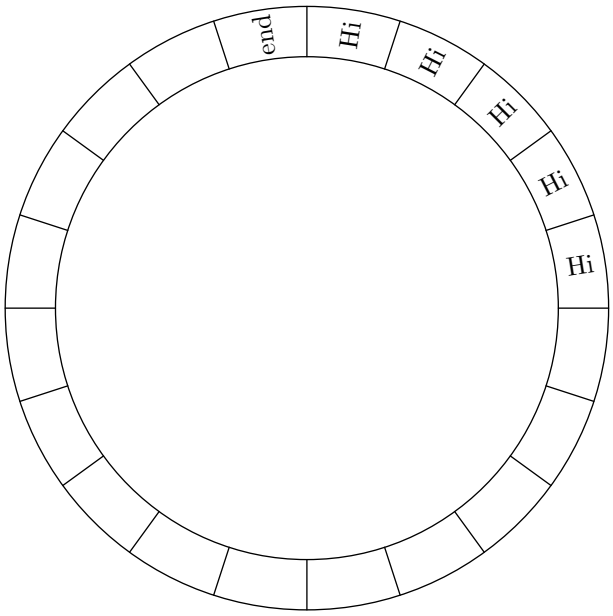
It remains to comment that the stack is a last in, first out structure, or **LIFO** structure.

§2.3 Queue

A **queue** is another ADT that has similar operations to a stack, but functions in a different way. In particular, it is a **FIFO** structure: first in, first out. We want a queue to support the following operations:

- `create`
- `is_empty`
- `enqueue`
- `dequeue` (remove oldest element)
- `front` (show oldest element)

A queue is somewhat more complicated to implement than a stack. If we use the same array-based approach, then we would have to keep track of both the front and back of the queue (since we remove from the front but add at the back). However, there is a way to remain efficient while doing this, by using a **circular array**.



With an array implementation, enqueue and dequeue are $O(1)$ operations.

Without further ado, we delve into the code. Here is C++ 's implementation of the queue:

```
#include <queue>

queue<int> q;
q.push(5);
```

```
q.push(3);
q.push(7);
cout << q.front() << "\n"; // 7
cout << q.back() << "\n"; // 5
q.pop();
cout << q.front() << "\n"; // 3
```

Here is an array implementation:

```
template<typename T>
class Queue {
public:
    void enQ(const T& x) {
        if (full()) {
            grow();
        }
    }
    void grow() { // Double size of array
        T *p = new T[2*(end - a)];
        // Copy elements from a to p
        // Update f,b
        delete[] a;
        a = p;
        end = m;
    }
}
```

It is important to note that doubling the size of the array when we grow ensures that (asymptotically) the expansion only takes as much time as actually inserting each element.

§2.4 Deque

A **deque** is a double-ended queue that supports the following operations:

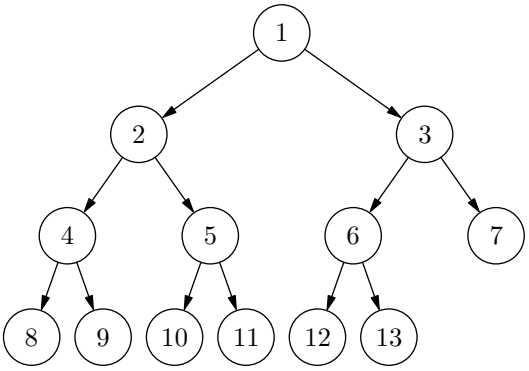
- `push_front`
- `push_back`
- `pop_front`
- `pop_back`
- `is_empty`

All of these should take $O(1)$ time.
Deques are implemented in exactly the same way as queues.

§2.5 Binary Trees

§2.5.1 Introduction

Consider the following example:



Node 1 would be called the **root** of the tree. Node 2 is the **parent** of nodes 4 and 5, but it is also the **child** of node 1. A node can have a left or right child.

While most nodes are parents and children, a node that does not have any children is called a **leaf**. In the representation above, nodes $8, 9, \ldots, 13$ are leaves of the tree. The **height** of this tree would be 3 since there are three "levels" of nodes.

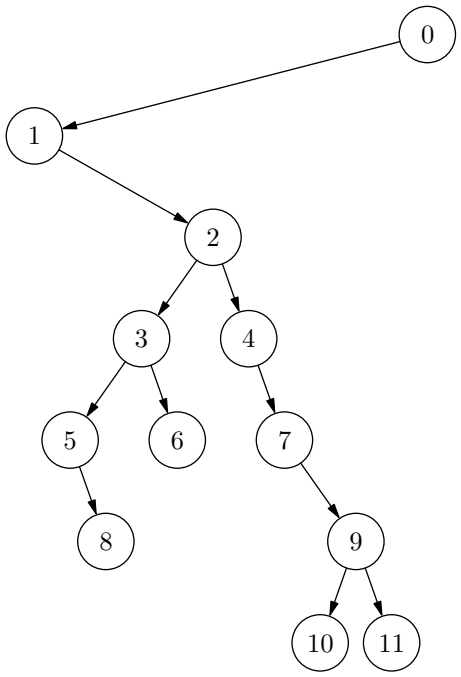
A **complete binary tree** is a binary tree where each level except the lowest is full.

Here is an implementation of the binary tree:

```
template<typename T>
struct BN {
    T key;
    int sz;
    BN* left;
    BN* right;
};

template<typename T>
int height(BN<T> *root) {
    if (root) return 1+ max(height(root->l),height(root->r));
    return -1;
}

template<typename T>
int size(BN<T> *root) {
    if (root) return 1+size(root->l)+size(root->r);
    return 0;
}

template<typename T>
BN<T>* find(BN<T>* root, int ord) {
    if (ord >= root->sz) return nullptr;
    int curr;
    while (curr != ord) {
        if (curr < ord) // update curr, root
            root = root->r;
            curr += 1 + (root->l ? root->l->sz : 0);
        else { // (curr > ord)
            // update curr, root
            root = root->l;
            curr -= 1+(root -> r?root->r->sz : 0);
        }
    }
    return root;
}
```

§2.5.2 Big Tree Traversal

Consider visiting the nodes in a tree in the following order:



Here is the implementation for this particular traversal:

```
void level(const BT* root) {
    queue<const BT*> q;
    q.push(root);
    while (!q.empty()) {
        root = q.front(); q.pop();
        visit(root); // do something to root
        if(root->l)
            q.push(root->l);
        if(root->r)
            q.push(root->r);
    }
}
```

The order that the nodes are visited is the level order. We have other ways of traversing the tree as well.

1. **In-order** traversal visits the left node, then root node, then right node.

```
    void in_order (const BT* root) {
        if (root) {
            in_order(root->l);
            visit(root);
            in_order(root->r);
        }
    }
```

2. **Pre-order** traversal visits the root node, then left node, then right node.

```
    void pre_order (const BT* root) {
        if (root) {
            visit(root);
            pre_order(root->l);
```

```
            pre_order(root->r);
        }
    }
```

3. **Post-order** traversal visits the left node, then right node, then root node.

```
    void post_order (const BT* root) {
        if (root) {
            post_order(root->l);
            post_order(root->r);
            visit(root);
        }
    }
```

Here is an implementation for **coastline** traversal:

```
void coastline(const BT* root) {
    if (root) {
        visit(root);
        if (root->l) {
            coastline(root->l);
            visit(root);
        }
        if (root->r) {
            coastline(root->r)
            visit(root);
        }
    }
}
```

§2.5.3 Binary Search Tree

Say that we store some *item* in each node, and the items have an *ordering*.

A binary tree is a **search tree** if an in-order traversal visits in order. This is one of the most common uses for binary trees.

If we traversed a search tree in-order, then we would get a horribly balanced tree in which its height would be the number of words. It would usually take $O(\log n)$ time to search for an element in a well balanced search tree.

§2.5.4 Containers and Iterators

An **iterator** is anything that can traverse items in a container.

For example, a pointer is an iterator:

```
++p; // p is a forward iter (Fiter)
--p; // p is a reverse iter (Riter)) % trivial parity argument
p += n; p -= n; // p is a random-access iter (RAiter)
```

§2.5.5 Counting

Example 2.2

How many binary trees are there which have n nodes?

Let $B(n)$ equal to the number of binary trees having n nodes. For our first few cases, we can enumerate $B(1) = 1$, $B(2) = 2$, and $B(3) = 5$. To find a formula in general, we should first find a recursion for $B(n)$.

Consider a root with at most two possible subtrees for its children. Let its left child have k nodes, and therefore its right child will have $n-k-1$ nodes. Note that a different value of k will yield a different tree. Then, for a particular k, there are $B(k) \cdot B(n-k-1)$ different possible trees when you consider a left and/or right subtree attached to a root node. Thus, the number of all binary trees with n nodes would be the summation from $k = 0$ to $n-1$, i.e.

$$B(n) = \sum_{k=0}^{n-1} B(k) \cdot B(n-k-1).$$

For consistency, define $B(0) = B(1) = 1$.

To solve this recurrence relation, we use **generating functions**. Let $B(n) = b_n$. Given a sequence $\{b_n\}$, its generating function

$$G(z) = \sum_{n=0}^{\infty} b_n z^n.$$

So, in our case,

$$G(z) = 1 + z + 2z^2 + 5z^3 + \cdots$$

Note that

$$\begin{aligned} G(z)^2 &= \left(\sum_{m=0}^{\infty} b_m z^m\right)\left(\sum_{n=0}^{\infty} b_n z^n\right) \\ &= b_0^2 + (2b_0b_1)z + (2b_0b_2 + b_1b_1)z^2 + (2b_0b_3 + 2b_1b_2)z^3 + \cdots \\ &= \sum_{l=0}^{\infty} z^l \sum_{k=0}^{l} b_k b_{l-k} \\ &= \sum_{l=0}^{\infty} z^l b_{l+1}, \end{aligned}$$

so $zG(z)^2 = z + 2z^2 + 5z^3 + \cdots$, hence

$$G(z) = 1 + zG(z)^2.$$

This is a quadratic equation in $G(z)$, namely

$$zG(z)^2 - G(z) + 1 = 0.$$

Then

$$G(z) = \frac{1 \pm \sqrt{1-4z}}{2z},$$

so we need to choose the proper sign. Note that $G(0) = 1$, and

$$\lim_{z\to 0} \frac{1-\sqrt{1-4z}}{2z} = \lim_{z\to 0} \frac{2}{1+\sqrt{1-4z}} = 1,$$

so we choose the negative sign (the positive sign would not make sense by this reasoning).

Now, given $G(z) = \frac{1-\sqrt{1-4z}}{2z}$, we need to find its coefficients.

We know that

$$\sqrt{1-4z} = (1-4z)^{1/2} = \sum_{n=0}^{\infty} a_n z^n$$

for some coefficients $a_0, a_1, \cdots$

Recall that

$$(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k,$$

where

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\cdots(n-k+1)}{k!}.$$

Newton's **Generalized Binomial Theorem** tells us that the formula above generalizes for real numbers, i.e.

$$\binom{\alpha}{k} = \frac{\alpha \cdot (\alpha-1)\cdots(\alpha-k+1)}{k!}$$

for $\alpha \in \mathbb{R}$.

Then,

$$\sqrt{1+x} = \sum_{k=0}^{\infty} \binom{1/2}{k} x^k,$$

where

$$\binom{1/2}{k} = \frac{(1/2)(1/2-1)\cdots(1/2-k-1)}{k!}.$$

We replace x with $-4z$ to obtain

$$\sqrt{1-4z} = \sum_{k=0}^{\infty} \binom{1/2}{k} (-4z)^k = \sum_{k=0}^{\infty} (-4)^k \binom{1/2}{k} z^k.$$

Thus,

$$\begin{aligned} G(z) &= \frac{1 - \sum_{k=0}^{\infty} (-4)^k \binom{1/2}{k} z^k}{2z} \\ &= \frac{1}{8} \sum_{k=1}^{\infty} (-4)^k \binom{1/2}{k} z^{k-1} \\ &= \frac{1}{8} \sum_{k=0}^{\infty} (-4)^{k+2} \binom{1/2}{k+1} z^k, \end{aligned}$$

but this is equal to

$$G(z) = \sum_{k=0}^{\infty} b_k z^k,$$

therefore

$$b_k = \frac{1}{8}(-4)^{k+2} \binom{1/2}{k+1}.$$

What is $\binom{1/2}{k+1}$? For the first few k, we have

$$\begin{aligned} \binom{1/2}{1} &= \frac{1}{2}, \\ \binom{1/2}{2} &= \frac{1/2 \cdot (1/2-1)}{2} = -\frac{1}{8}, \end{aligned}$$

$$\binom{1/2}{3} = \frac{1/2 \cdot (1/2-1) \cdot (1/2-2)}{6} = \frac{1}{32}.$$

We observe that

$$\binom{1/2}{k} = \frac{1}{2^k k!} \prod_{j=1}^{k} (-2j+3),$$

and

$$\prod_{j=1}^{k} (-2j+3) = (-1)^{k+1} \frac{(2k-3)!}{2^{k-2}(k-2)!},$$

so

$$\begin{aligned}
\binom{1/2}{k} &= \frac{1}{2^k k!} \prod_{j=1}^{k} (-2j+3) \\
&= \frac{1}{2^k k!} (-1)^{k+1} \frac{(2k-3)!}{2^{k-2}(k-2)!} \\
&= \frac{(-1)^{k+1}(2k-3)!}{2^{2k-2} k!(k-2)!} \\
&= \frac{(-1)^{k+1}(2k-2)!}{2^{2k-2}(2k-2) k!(k-2)!} \\
&= \frac{(-1)^{k+1}(2k-2)!}{2^{2k-1} k!(k-1)!} \\
&= \frac{(-1)^{k+1}}{2^{2k-1} k} \binom{2k-2}{k-1}.
\end{aligned}$$

Therefore,

$$\begin{aligned}
b_k &= \frac{1}{8}(-4)^{k+2} \frac{(-1)^{k+2}}{2^{2k+1}(k+1)} \binom{2k}{k} \\
&= \frac{1}{k+1} \binom{2k}{k}.
\end{aligned}$$

How large is this? By Stirling's Approximation formula,

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n.$$

Then

$$\binom{2n}{n} = \frac{(2n)!}{(n!)^2} \sim \frac{\sqrt{4\pi n} \left(\frac{2n}{e}\right)^{2n}}{2\pi n \left(\frac{n}{e}\right)^{2n}} = \frac{4^n}{\sqrt{\pi n}}.$$

Therefore, $b_n \sim \frac{4^n}{(n+1)\sqrt{\pi n}}$, and is certainly exponential in growth. A special name for the sequence $\{b_n\}$ is the **Catalan numbers**.

As a recap of what happened:

1. $B_n = \sum_{k=0}^{n-1} B_k B_{n-k-1}$
2. We defined the generating function $G(z) = B_0 + B_1 z + B_2 z^2 + \cdots$
3. We obtain the equation $zG(z)^2 = G(z) - 1$, and solved to get $G(z) = \frac{1-\sqrt{1-4z}}{2z}$.

4. Using the Taylor expansion (i.e. $(1+x)^{1/2} = \sum_{n=0}^{\infty} \binom{1/2}{n} x^n$), we get the closed form $B_n = \frac{1}{n+1}\binom{2n}{n}$.

As another example using generating functions, consider the **Fibonacci numbers**, where we define $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$.

Example 2.3
Let $f(z) = \sum_{n=0}^{\infty} F_n z^n$. Get an equation for f, find f explicitly, and use f to get a formula for F_n.

§2.5.6 Binary Tree Deletion

We have three cases to consider:

a) The node is a leaf.

```
void remove(const K& key, BT *root) {
    delete dad->r; // or dad->l
    dad->r = nullptr;
}
```

b) Node has one child

c) Node has 2 children

Make a choice: rightmost node on Node's left subtree, or leftmost of a right subtree.

- `r` has ≤ 1 child. By case (a) or (b), remove (but don't delete `r`). Make `r->l` and `r->r` point to `p->l` and `p->r` respectively. Then make `dad->r=r`, and `delete p`.

  This is O(tree height), not an $O(1)$ operation.

§2.6 Binary Tree Iterators

Ideally, we want the following code to work:

```
BT *root = get_tree();
auto it = root->begin();
while (it != root->end())
   visit(*it++); // In-order traversal
```

How do we implement this?

1. We can keep a stack of ancestors with each iterator. Then the space is O(height).
2. Store parent pointer in each node. Then space is $O(n)$.

§2.7 Balanced Trees

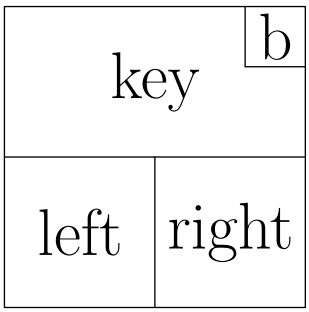
Two nodes are **siblings** if their parents are the same.

A binary tree is **balanced** if the heights of any two sibling subtrees do not differ too much. We want the height to be $O(\lg n)$, because then searching the tree would be efficient. There are many ways to accomplish this:

1. AVL trees: heights differ by ≤ 1

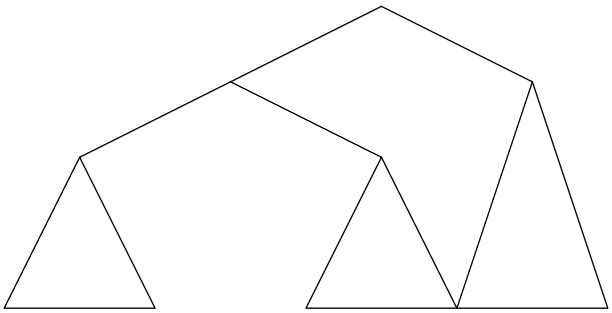
2. Red-Black Trees
3. Splay Trees
4. Treaps
5. α-trees

To implement an AVL tree, we consider a node with an additional property:



The balancing factor b is in the set $\{-1, 0, 1\}$, where it is equal to the height of the right tree minus the height of the left tree.

To completely insert a node, we also have to rebalance with tree rotations.

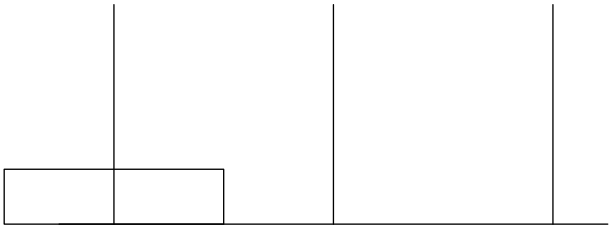


```
// Return b (O(1))
BT* rr(BT *c) {
    if (c) {
        BT* b = c->l;
        c->l = b->r;
        b->r = c;
        return b;
    }
    return c;
}
```

In an AVL tree of height h, what is the minimum number of nodes? Maximum number of nodes?

Let M_h be the minimum number of nodes. Then $M_h = 1 + M_{h-1} + M_{h-2}$, where we define $M_0 = 1$ and $M_1 = 2$. This is because we count the root itself as a node, then consider the left/right subtrees having heights of $h - 1$ and $h - 2$ (at worst).

This recurrence is closely related to the Fibonacci numbers - in fact, since we defined $M_0 = 1$ and $M_1 = 2$, $M_h = F_{h+3} - 1$.



§2.8 Recursion

Recall the classic **Towers of Hanoi** problem:

The aim is to transfer disks so that no larger disk is on top of a smaller disk (one at a time).

```
1 void hanoi(int n, int f, int t, int x) { // from, to, free
2     if (n == 0) return;
3     hanoi(n-1,f,x,t);
4     cout << f << " ==> " << t << "\n";
5     hanoi(n-1,x,t,f);
6 }
```

Now consider the variant problem **Circular Hanoi**, in which we have the same three towers except that you are only allowed to move from tower 1 to tower 2, tower 2 to tower 3, and tower 3 back to tower 1.

Let f_n be the number of moves to transfer n disks to non-adjacent tower, and c_n be the number of moves to transfer n disks to the adjacent tower. Then we compute

$$\begin{aligned} f_n &= 2f_{n-1} + 2f_{n-2} + 3, \\ f_0 &= 0, \\ f_1 &= 2, \\ c_n &= 2f_{n-1} + 1. \end{aligned}$$

Example 2.4

Suppose we have a linear, homogeneous, and constant coefficient second order recurrence:

$$a_n = Aa_{n-1} + Ba_{n-2}.$$

Guess $a_n = r^n$. Then

$$\begin{aligned} r^n &= Ar^{n-1} + Br^{n-2} \\ r^2 &= Ar + B \\ r^2 - Ar - B &= 0 \end{aligned}$$

By the quadratic formula,

$$\begin{aligned} r_1 &= \frac{A + \sqrt{A^2 + 4B}}{2}, \\ r_2 &= \frac{A - \sqrt{A^2 + 4B}}{2} \end{aligned}$$

Then $\forall c_1, c_2$,

$$a_n = c_1 r_1^n + c_2 r_2^n$$

is a solution.

Let $a_0 = \alpha$ and $a_1 = \beta$. Then solve

$$\begin{aligned} c_1 + c_2 &= \alpha, \\ c_1 r_1 + c_2 r_2 &= \beta. \end{aligned}$$

In the particular example of Circular Hanoi, we have the **indicial equation** (ignoring the constant term at first):

$$x^n = 2x^{n-1} + 2x^{n-2},$$

and dividing by x^{n-2} gives

$$x^2 - 2x - 2 = 0.$$

This has roots $r_1 = 2 + \sqrt{3}$, $r_2 = 2 - \sqrt{3}$. Then our general solution of the homogenous recurrence is

$$c_1 r_1^n + c_2 r_2^n.$$

Let $f_n = g_n + \alpha$. Then our recurrence becomes

$$g_n + \alpha = 2(g_{n-1} + \alpha) + 2(g_{n-2} + \alpha) + 3,$$

which rearranges to

$$g_n = 2g_{n-1} + 2g_{n-2} + 3(\alpha + 1),$$

and we want the constant term to disappear. To make this happen, let $\alpha = -1$, i.e. let $f_n = g_n - 1$. Then based on our indicial equation we had just solved, we have

$$g_n = c_1 r_1^n + c_2 r_2^n.$$

Then $g_0 = c_1 + c_2 = 1$, and $g_1 = c_1(1 - \sqrt{3}) + c_2(1 + \sqrt{3}) = 3$. Solving for c_1 and c_2, we ultimately get the recurrence

$$g_n = \frac{1}{2}\left(1 - \frac{2}{\sqrt{3}}\right)\left(1 - \sqrt{3}\right)^n + \frac{1}{2}\left(1 + \frac{2}{\sqrt{3}}\right)\left(1 + \sqrt{3}\right)^n,$$

hence

$$f_n = \frac{1}{2}\left(1 - \frac{2}{\sqrt{3}}\right)\left(1 - \sqrt{3}\right)^n + \frac{1}{2}\left(1 + \frac{2}{\sqrt{3}}\right)\left(1 + \sqrt{3}\right)^n - 1.$$

Noting that $\left|1 - \sqrt{3}\right| < 1$, the term $\frac{1}{2}\left(1 - \frac{2}{\sqrt{3}}\right)\left(1 - \sqrt{3}\right)^n$ becomes negligible for large n. Hence, the growth of f_n is dominated by the $\frac{1}{2}\left(1 + \frac{2}{\sqrt{3}}\right)\left(1 + \sqrt{3}\right)^n$ term, so it has a growth of $O((1 + \sqrt{3})^n)$.

Example 2.5

Consider the Towers of Hanoi problem for more than 3 towers. Define $H(d, t)$ as the number of moves it takes to move d disks among t towers.

1. Choose some k.
2. Move k disks to the spare tower, which takes $H(k, t)$ moves.
3. Move the rest to the goal tower, which takes $H(n - k, t - 1)$ moves.
4. Move the k disks back to the goal tower, which takes another $H(k, t)$ moves.

Thus the total number of moves is $H(n, t) = H(n - k, t - 1) + 2H(k, t)$.
To find the optimal value of k, we minimize over all possible values of k:

$$H(n, t) = \min_{1 \le k \le n-1} \{H(n - k, t - 1) + 2H(k, t)\},$$

using dynamic programming.

§3 Asymptotic Analysis

We seek to classify algorithms by time and space requirements.

Size can be associated with an instance of the algorithm. For example, in sorting, the "size" of the input could be considered the number of input elements (although this is an abstraction, since we are not specifying the size of the items).

As a function of n, how is the fast is the algorithm? How much space does it require? For each of these questions, it is important for us to consider:

- Average case
- Best case
- Worst case

Sometimes, the average case is hard to compute, especially without discussing probability distributions. The worst case can be unnecessarily pessimistic.

Now, how do we specify what units we use when we describe time requirements? We can't use "seconds" or "number of instructions" because that is specific to your machine, and not a property of the algorithm. We usually specify a higher level operation, and measure how many times that operation is used. For example, in sorting, we would use "number of comparisons," or the "number of swaps."

Therefore, for a given application, what do we count? It should be platform and language independent, meaning that it doesn't matter whether you're running the algorithm on your own laptop or a supercomputer, or using Java or Python. Furthermore, how do we express such a number?

Space is the number of bytes. Again, we consider: How much space does the algorithm require? But this brings us to the same issue: what do we count?

To gain, let $T_1(n) = 2.1n^2 - 3n + 17$ and $T_2(n) = 2.5n^2 - 53n - 1000$ for two algorithms A_1 and A_2 respectively. For small n, T_2 is faster than T_1, but for large n, T_1 is faster (just look at the leading terms):

$$\lim_{n\to\infty} \frac{T_1(n)}{T_2(n)} = \frac{2.1}{2.5} < 1.$$

Definition 3.1. $f(n) \sim g(n)$ (read as "$f(n)$ is **asymptotic** to $g(n)$") if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

A famous example in mathematics is the Prime Number Theorem. Let $\pi(n)$ be the number of positive primes less than or equal to n. For instance, $\pi(10) = 4$ and $\pi(100) = 25$. Then,

$$\pi(n) \sim \frac{n}{\ln n}.$$

Specifically, mathematician Pafnuty Chebyshev proved that if $\lim_{n\to\infty} \frac{\pi(n)}{\left(\frac{n}{\ln n}\right)}$ exists, it is 1. It was proven later that the limit does in fact exist.

Definition 3.2. We say $f(n) = O(g(n))$ if

$$\exists M > 0 : \lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < M$$

In other words, for sufficiently large n,

$$|f(n)| \leq M|g(n)|.$$

In computer science, we usually drop the absolute values since everything is positive.

For example, $T_1(n) = 2.1n^2 - 3n + 17 = O(n^2)$, because for large n, $T_1(n) < 3n^2$.

It is important to realize that $T_1(n)$ is also $O(2^n)$, since all we care about is that f is smaller than g.

Definition 3.3. $f(n) = \Theta(g(n))$ if

$$\exists M_1, M_2 > 0 : \text{for sufficiently large } n,\ M_1|g(n)| \leq |f(n)| \leq M_2|g(n)|.$$

For example, $T_1(n) = \Theta(n^2)$ but $T_1(n) \neq \Theta(n^3)$.

Definition 3.4. $f(n) = \Omega(g(n))$ if for sufficiently large n,

$$\exists M > 0 : M|g(n)| \leq |f(n)|.$$

Definition 3.5. $f(n) = o(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

For example, $T_1(n) = o(n^3)$.

Definition 3.6. $\lg^*(n)$ is the greatest number of lgs required so that

$$\underbrace{\lg \lg \lg \lg \cdots \lg \lg \lg(n)}_{\lg^*(n)} < 1.$$

Problem 3.7. Rank by order of growth. If $f(n) = \Theta(g(n))$ they have same rank:

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$\left(\frac{3}{2}\right)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{\frac{1}{\lg n}}$
$\ln \ln n$	$\lg^* n$	$n2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

Solution.

1. $n^{\lg n}$
2. $\lg(\lg^* n)$
3. $\lg^*(\lg n)$, $\lg * n$
4. $2^{\lg^* n}$
5. $\ln \ln n$
6. $\sqrt{\lg n}$
7. $\ln n$
8. $\lg^2 n$

   Note that this is the same as $2^{2 \lg \lg n}$.

9. $2^{\sqrt{2 \lg n}}$
10. $\sqrt{2}^{\lg n}$
11. n, $2^{\lg n}$
12. $n \lg n$, $\lg n!$

    Notice that $\lg n! = \sum_{k=2}^{n} \lg k = \Theta\left(\int_1^n \lg x \, dx\right)$. Furthermore, **Stirling's Approximation** notes that $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.
13. n^2, $4^{\lg n}$
14. n^3
15. $(\lg n)!$

    As a fun fact, here is the **Gamma Function**:

    $$\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} \, dt,$$

    from which it follows from integration by parts that $\Gamma(x+1) = x\Gamma(x)$ and more generally $\Gamma(n) = (n-1)!$.
16. $(\lg n)^{\lg n}$, $n^{\lg \lg n} = (2^{\lg n})^{\lg \lg n}$
17. $\left(\frac{3}{2}\right)^n$
18. 2^n
19. $n2^n$
20. e^n
21. $n!$
22. $(n+1)!$
23. 2^{2^n}
24. $2^{2^{n+1}}$

□

§4 Algorithms

§4.1 Sorting

§4.1.1 Insertion Sort

Given an array of numbers $a_0, a_1, \ldots, a_n$, we want to return a permutation of this array such that $a_{p_0} \leq a_{p_1} \leq \cdots \leq a_{p_n}$.

One possible approach is **insertion sort**.

```
#include <iostream>
#include <vector>
using std::vector;
using std::cout;

template<typename T>
void insertion_sort(vector<T>& a) {
    for (int i = 1; i < a.size(); ++i) {
        int t = i;
        while (t > 0 && a[t] < a[t-1]) {
            int s = a[t];
            a[t] = a[t-1];
            a[t-1] = s;
            t--;
        }
    }
}
```

The worst case comes from sorting a completely reversed array. For $i = 1, \ldots, n-1$ where n is the size of the array, we make i comparisons, so our total number of comparisons is

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n-2)}{2} \approx \frac{n^2}{2},$$

so our algorithm is $\theta(n^2)$.

The best case comes from sorting an already sorted array. Then, we will only make one comparison at each $i = 1, 2, \ldots, n-1$, so our total number of comparisons is

$$\sum_{j=1}^{n-1} 1 = n - 1,$$

giving an $O(n)$ best case.

The average case is approximately $\frac{n^2}{4}$.

So, we can see that if the list is "close to sorted," then the algorithm is "close to O(n)." There are some reasons to use insertion sort:

1. As a final cleanup after more complicated results.
2. For small n, insertion sort has a low complexity: $O(n^2) < O(n \log n)$ for small n.

Bubble sort is useless: it is twice as slow as average case of insertion, and it is never $O(n)$ - it is always $\theta(n^2)$.

§4.1.2 Shell Sort

§4.1.3 Inversions

Given an array

$$a_0, a_1, \ldots, a_{n-1},$$

an **inversion** is an ordered pair (i, j) such that $i < j$ and $a_i > a_j$. Swapping 2 adjacent elements decreases the number of inversions by ≤ 1. Then

$$0 \leq \text{number of inversions} \leq \binom{n}{2}.$$

This means that we can have $\theta(n^2)$ inversions.

Any algorithm that only swaps adjacent elements has worst case $\theta(n^2)$.

Inversions can be counted in $O(n \log n)$ time using a variant of merge sort.

§4.1.4 Lower Bounds

We wish to find a lower bound for comparison-based sorting (i.e. sorting where you can only ask whether one element is less than another).

Suppose we are given distinct elements, and consider a decision tree. It will be a binary tree such that at each node, we make the decision to go left (which represents $<$) or right (which represents $>$). Then we end up with a binary tree such that the leaves of the nodes signify that the array is now sorted.

Then, we want to minimize the number of comparison-steps taken, i.e. the height of this decision tree. Let d be the number of steps. Then there will be at most 2^d leaves. Let there be n elements. Then this decision tree has to distinguish between every possible permutation of these n elements, and there are $n!$ arrangements.

Then, we necessarily have $2^d \geq n!$, i.e. $d \geq \log(n!)$. Then $d \geq n \ln n$, our lower bound. In practice, the best time complexity for sorting is $O(n \log n)$.

Definition 4.1. A sort is **stable** if it preserves the relative order of equal elements.

In other words, if $a_i = a_j$ for $i < j$ initially, then at the end of the sort, $\ldots a_{\pi_i} \ldots a_{\pi_j} \ldots$, $\pi_i < \pi_j$ after sorting.

§4.1.5 Merge Sort

The downside of this sorting method is that it uses twice the storage (a whole copy of the array that is being sorted). Sorting n elements require $2n + \epsilon$ where ϵ is $O(1)$ bookkeeping.

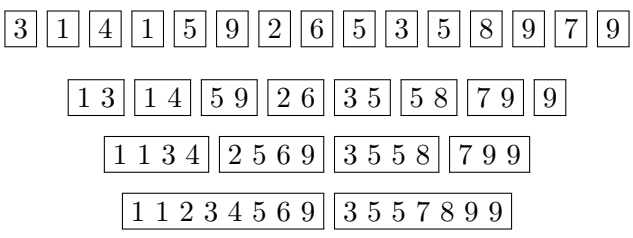
We can go about implementing this in two ways: top-down or bottom-up. First consider bottom-up.

1. Sort $\{a_{2i}, a_{2i+1}\}$, $i = 0, 1, 2, \ldots, \lfloor \frac{n}{2} \rfloor$.
2. Merge $\{a_{4i}, a_{4i+1}\}$, $\{a_{4i+2}, a_{4i+3}\}$, for $i = 0, 1, \ldots, \lfloor \frac{n}{4} \rfloor$.

We keep merging sub-arrays of size 2^k, $k = 1, 2, \ldots, \log n$.

At some k, the merging itself takes $O(n)$ time. Then, the overall time complexity is $O(kn) = O(n \log n)$ since k goes up to $\log n$.

Here is bottom-up merge sort performed on the array $\{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9\}$:



1 1 2 3 3 4 5 5 5 6 7 8 9 9 9

We can turn any sort into a stable sort by augmenting the keys and $<$.

Merge sort can also be implemented top-down. First, given some array $\{a_1, a_2, \ldots, a_n\}$, we split it into $\{a_1, a_2, \ldots, a_{\lfloor n/2 \rfloor}\}$ and $\{a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n\}$. Then, apply merge sort recursively on to each half. Lastly, we merge the 2 (now sorted) pieces together.

It turns out that linked lists are perfect for applying merge sort, since we can easily "cut" the list into two halves by adjusting the `next` pointer, and declaring a new array is not necessary. Here is an implementation for top-down merge sort with a linked list:

```
template<typename K>
struct Link {
    K key;
    Link<K>* next;
}

template<typename K>
Link<K>* merge_sort(Link<K> *head) {
    if (head->next == nullptr) return head; // A single element list is already sorted
    // First, count size of list
    int size = 0;
    Link<K> *cur = head;
    while (cur != nullptr) {
        ++size;
        cur = cur->next;
    }
    // Take first half
    int half = size/2-1;
    cur = head;
    int count = 0;
    while (count < half) {
        count++;
        cur = cur->next;
    }
    // Cut off second half
    Link<K> *second_half = cur->next;
    cur->next = nullptr;
    // Recursively apply merge sort
    Link<K> *fp = merge_sort(head);
    Link<K> *sp = merge_sort(second_half);

    // Now merge the two lists, then return
    Link<K> *new_merge;
    if (fp->key < sp->key) {
        new_merge = fp;
        fp = fp->next;
    } else {
        new_merge = sp;
        sp = sp->next;
    }
    Link<K> *new_head = new_merge;
    while (fp != nullptr && sp != nullptr) {
        if (fp->key < sp->key) {
            new_merge->next = fp;
            new_merge = fp;
            fp = fp->next;
        } else {
            new_merge->next = sp;
            new_merge = sp;
            sp = sp->next;
        }
```

```
    }
    while (fp != nullptr) {
        new_merge->next = fp;
        new_merge = fp;
        fp = fp->next;
    }
    while (sp != nullptr) {
        new_merge->next = sp;
        new_merge = sp;
        sp = sp->next;
    }
    return new_head;
}
```

§4.1.6 Radix Sort (Bucket Sort)

The given key is a sequence of values where each value takes on a few discrete values. If a given array has terms only in the set of keys $\{a_1, \ldots, a_n\}$, then by counting how much of each term appears in the array, the sorted array can be "constructed" in $O(n)$ time.

For example, if we had the unsorted array $\{3, 2, 2, 1, 3, 3, 1, 1, 2, 1, 3\}$ and we knew beforehand that each element was one of $1, 2$, or 3, then we can iterate through the array once to count that there are three "buckets:" 4 1's, 3 2's, and 4 3's in total. Then we can immediately construct the array by placing 4 1's in the beginning, followed by 3 2's and then 4 3's, yielding $\{1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3\}$ as the sorted array.

However, bucket sort is only efficient when the keys are uniformly distributed over a range. Some keys are less obvious (such as the least or most significant bit) Some buckets could contain more elements than average; if all elements are placed into the same bucket, then the runtime is $O(n^2)$.

§4.1.7 Quick Sort

Quicksort is possibly the best sorting algorithm. First we discuss the basic mechanism, and then introduce multiple ways of optimizing it. Here's how it works: let the array be called a.

- Choose a partition element x (How? TBD.) In **vanilla quicksort**, choose x to be the first element of the array.
- Partition the array, so that every y in a where $y \leq x$ ends up to the left of x, and if $y \geq x$, y ends up to the right of x.
- Recursively quicksort the right and left subarrays.

```
void quick_sortV(int a[], int n) {
    qs1(a, 0, n-1);
}

void qs1(int a[], int lo, int hi) {
    if(lo < hi) {
        int m = qslPartition(a, lo, hi);
        qs1(a, lo, m-1);
        qs1(a, m, hi);
    }
}

int qs1Partition(int a[], int lo, int hi) {
```

```
    int x = a[lo]; // Vanilla choice of partition elements
    while(lo <= hi) { // Welcome to the famous inner loops of quicksort
        while(a[lo] < x) ++lo;
        while(a[hi] > x) --hi;
        if (lo <= hi) {
            int t = a[lo];
            a[lo++] = a[hi];
            a[hi--] = t;
        }
    }
    return lo;
}
```

If the array is big enough, quicksort might crash because of the recursion limit. If we happen to choose a pivot that partitions the array into a subarray of 1 element and another of $n-1$ elements, then when we recursively call quicksort on the latter, the recursion depth can be $n-1$. Then an embarrassing problem is that if the array is close to sorted, quicksort runs in $\theta(n^2)$ time.

That ain't good. To remedy this issue, we introduce **strawberry quicksort** (a few modifications). The main idea is to recurse only on the shorter piece. Then the depth of the recursion is $O(\lg n)$.

```
// Instead of qs1
void qs2 (int a[], int lo, int hi) {
    while (lo < hi) {
        int m = qslPartition(a,lo,hi);
        if (m - lo < hi - m + 1) {
            qs2(a, lo, m - 1);
            lo = m;
        } else {
            qs2(a, m, hi);
            hi = m - 1;
        }
    }
}
```

However, our worst case is still $\theta(n^2)$ given an array already sorted.

But wait, there's more! There's another slight modification, now dubbed **coffee quick sort**, that helps us practically eliminate $\theta(n^2)$ behavior. We change one line in the implementation of the partition function:

```
int qs3Partition(int a[], int lo, int hi) {
    int x = median_of_3 (a,lo,hi);

    // Rest of code is the same
}
```

By `median_of_3`, we mean that if we let `int med = (lo+hi)/2`, then choose x to be the median of the set {`a[lo]`,`a[med]`,`a[hi]`}.

Finally, we have **chocolate quicksort**. The idea is that short arrays (length less than a certain, specified `THRESHOLD`) can be sorted more quickly by a simple method.

```
void qs4 (int a[], int lo, int hi) {
    while (hi - lo > THRESHOLD)
        // same as qs2
    // After this, we are left with a bunch of small, unsorted arrays
}

void quicksort(int a[], int n) {
    qs4(a,0,n-1);
    // We use insertion sort to sort the remaining small arrays
```

```
10      insertion_sort(a,n);
11  }
```

We take advantage of the fact that insertion sort is faster than standard quicksort on small arrays.

§4.1.8 Finding the Median (Order Statistics)

We want to find the k-th smallest element of an array. We use a modified quicksort:

- Partition the array as in quicksort
- Recurse on the part that contains the k-th smallest element.

Here is some pseudo-code (although the array will be destroyed after you finish:

```
x = median_of_3(a,0,n-1);
m = partition(x,a);
if (k < m)
    find_kth(a,0,m-1,k);
else
    find_kth(a,m,n-1,k-m);
```

§4.1.9 Heapsort

A **heap** is a "full" binary tree where each element is greater than or equal to its children. ("Full" as in all but the children in bottom layer have two children). We will index the nodes in the tree using level-order. Using this numbering system, the left child of index i is $2i$, the right child has index $2i + 1$, and the parent has index $\lfloor \frac{i}{2} \rfloor$. This leads to the realization that we don't need to store this as a tree: we can just store it as an array!

So, how do we build the heap? It turns out that we can do it in $2n$ operations. Consider the following pseudocode:

```
x = largest index with children
for i from x to 1:
    t = i
    while heap[t] < heap[2*t] or heap[t] < heap[2*t+1]:
        swap heap[t] with its larger child
        update t to be index of larger child
```

Now, how do we pop off the biggest element (the root)?

```
n = size(heap)
swap heap[1] with heap[n], remove heap[n] from the heap
t = 1
while heap[t] < heap[2*t] or heap[t] < heap[2*t+1]:
    swap heap[t] with its larger child
    update t to be index of larger child
```

This is not very often used as a sorting algorithm, since it's worse than quicksort. However, it is very frequently used as a method to implement a **priority queue**. A priority queue supports the following three operations:

- Create an empty priority queue.
- Add an element with a certain "priority."
- Remove the element with highest priority.

To implement this, we can use the exact same heap from before. The only new operation is insertion. To do this, we add a new element at the first empty position in the array, and "bubble" it up the heap until it satisfies the heap property. This takes $O(\lg n)$.

§4.1.10 Sliding Maximum Window

§4.1.11 Summary

1. Mergesort: $\theta(n \lg n)$; stable
2. Quicksort: $\theta(n \lg n)$ (Can be $\theta(n^2)$)
3. Heapsort: $\theta(n \lg n)$
4. Insertion sort: $O(n^2)$ (Can be $O(n)$); stable
5. Radix sort: $O(kn)$ where there are k fields; possibly stable
6. Distribution Counting: $O(kn)$; possibly stable
7. Order Statistics: $O(n)$

External Sorting: Not enough memory to hold entire array in RAM
Parallel Sorting: Multiple Processors

§4.2 Convex Hulls

Now we delve into **computational geometry**, which leads to questions or queries such as:

- Is a point inside a polygon?
- What is the closest pair of points?
- What is the smallest graph connecting points in the plane?
- What point in a given set is closest to an arbitrary location?
- Is a sequence of line segments simple?
- What is the convex hull of a set of points?
- What is a "good triangulation"? (Delaunay triangulation)

Find the counterclockwise (CCW) traversal of points on the boundary of the convex hull.

First, we should do some preprocessing: form a "simple" shape (e.g. triangle or quadrilateral) and eliminate all points inside.

One algorithm we can use to find the convex hull is **Divide-and-Conquer**. After choosing "leftmost" and "rightmost" points in the set (which divide the set into "top" and "bottom" parts), we can recursively construct convex hulls for the top and bottom halves of the set and then merge the two together.

Another method we can use is **Graham-Scan**. First, if we set the bottom-left point in the set as the origin, then we sort the other points by the angle it forms with the bottom-left point with respect to the x-axis.

§4.3 Hashing

Suppose we want to store many pairs of keys and values, where the keys are not ordered. The main idea is that we want a unique location for each key.

Given $O(m)$ different keys in use, we set the location equal to the key modulo m, (preferably m is the size of the array). Now, collisions can occur: 2 distinct keys can map to the same location.

If we have n keys, then there are m^n different ways to map the keys to the values. If $n < m$, then are

$$\frac{m(m-1)\cdots(m-n+1)}{n!} = \binom{m}{n}$$

different maps that map keys to distinct values. But

$$\lim_{n\to\infty} \frac{\binom{m}{n}}{m^n} = 0,$$

at a very fast rate, so collisions are a necessary evil.

There are two methods to remedy this:

1. Chaining (more space): All keys that have equal hash values are on a linked list.
2. Open Addressing (no deletion)

Definition 4.2. The **load factor** is denoted by α, where

$$\alpha = \frac{n}{m},$$

where there are n keys in the table and m is the size of the table.

By definition, α is the average length of a chain (if we were using chaining).

The number of probes is $\frac{\alpha}{2}$ on average, so the total time is $1 + \frac{\alpha}{2}$, where we have to find the initial location.

Clearly $\alpha = \frac{n}{m} < 1$, so performance degrades drastically as $\alpha \to 1$. An unsuccessful search takes approximately $\frac{1}{1-\alpha}$ probes. When $\alpha = \frac{1}{2}$, the expected number of probes is 2. The choice of α is typically $\frac{3}{4}$.

Let $\mathcal{U}$ be the universe of key values. We can assume a map from $\{\text{Objects}\} \to \mathcal{U}$, where usually $\mathcal{U} \subset \mathbb{N}$.

First, we can turn an object into a string of bytes:

$$\left(\sum_{k=0}^{l-1} 31^k \cdot a[k]\right) \%\text{big number}.$$

1. Division Method: Let m be prime. THen define $h(x) = x\%m$. Do not use $m = 2^j$.
2. Multiplicative Method: Choose your favorite real number, such as $\phi = \frac{1+\sqrt{5}}{2}$ or π^e. Then let $h(k) = \lfloor m\left((kA) \mod 1\right)\rfloor$.

§4.3.1 Open Addressing Performance

What is the expected number of probes for unsuccessful search? Recall that the load factor $\alpha = \frac{n}{m}$, and that

$$P(i \text{ probes}) = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \left(1 - \frac{n-i+1}{n-i+1}\right).$$

The expected number of probes is

$$1 \cdot P(1) + 2 \cdot P(2) + \cdots = P(i > 0) + P(i > 1) + \cdots,$$

so

$$P(\#\text{ of probes } > i) = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1}.$$

Thus, the expected number is

$$\sum_{i=1}^{m-1} \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} \leq \sum_{i=1}^{\infty} \left(\frac{n}{m}\right)^i = \sum_{i=1}^{\infty} \alpha^i = \boxed{\frac{1}{1-\alpha}}.$$

For a successful search, we compute

$$\frac{1}{n} \sum_{k=1}^{n} (\#\text{ of probes to find the item inserted } k^{\text{th}}).$$

This is equal to

$$\begin{aligned}
\frac{1}{n} \sum_{k=0}^{n-1} \frac{1}{1-\alpha_{k+1}} &= \frac{1}{n} \sum_{k=0}^{n-1} \frac{1}{1-\frac{k}{m}} \\
&= \frac{1}{n} \sum_{k=0}^{n-1} \frac{m}{m-k} \\
&= \frac{m}{n} \sum_{k=0}^{n-1} \frac{1}{m-k} \\
&= \frac{m}{n} \left(\frac{1}{m} + \frac{1}{m-1} + \cdots + \frac{1}{m-n+1} \right) \\
&= \frac{1}{\alpha} \left(H_m - H_{m-n} \right).
\end{aligned}$$

Now, we use the well known fact that $H_m \sim \ln m$ (which can be observed by comparing the integral of $\frac{1}{x}$ by the Riemann sum), to obtain

$$\begin{aligned}
&= \frac{1}{\alpha} \left(\ln m - \ln(m-n) \right) \\
&= \frac{1}{\alpha} \ln \left(\frac{m}{m-n} \right) \\
&= \frac{1}{\alpha} \ln \left(\frac{1}{1-\frac{n}{-m}} \right) \\
&= \frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right).
\end{aligned}$$

§4.3.2 Improved Open Addressing

The key to this is **quadratic secondary hashing**.

§5 Graphs

§5.1 Terminology

Definition 5.1. A **graph** is a pair (V, E) where V is a finite set, and E is a multiset whose elements are of the multiset form $\{x, y\}$ where $x, y \in V$.

For example, let $V = \{\text{dog, clock, train, airport}\}$ and $E = \{\{\text{train, dog}\}, \{\text{clock,train}\}, \{\text{airport, airport}\}, \{\text{clock,airport}\}\}$. This is given by the visual representation

Definition 5.2. The set V is the **vertices**.

Definition 5.3. The set E is the **edges**.

Definition 5.4. The graph is **simple** if E is a set of subsets (i.e. no loops or multiple of the same edge).

Definition 5.5. A **directed graph**, or **digraph**, is a pair (V, E) where V is a finite set, and $E \subset V \times V$ where we note that $(x, y) \neq (y, x)$.

Definition 5.6. In a digraph, E is the set of **arcs**.

Let $G = (V, E)$, where $V = \{a, b, c\}$, $E = \{(a, b), (b, c), (c, a)\}$. Then

Definition 5.7. A **walk in** of $G = (V, E)$ is a sequence of vertices and edges

$$v_0 e_0 v_1 e_1 \dots e_n v_{n+1}$$

where $e_i = \{v_{i-1}, v_i\}$.

Definition 5.8. A **trail** is a walk in which edges are not repeated.

Definition 5.9. A **path** is a trail in which no vertices are repeated.

Definition 5.10. A **circuit** is a trail in which the first and last vertices are equal.

Definition 5.11. A **cycle** is a circuit in which no vertices except the first and last are repeated.

Definition 5.12. If an edge $e = \{x, y\}$, then the **endpoints** of e are x and y.

Definition 5.13. Given endpoints x and y of an edge e, we say x and y are **incident** on e.

Definition 5.14. The number of edges incident on a vertex x is the **degree** of x.

Definition 5.15. For digraphs, the **in-degree** of a vertex x is the number of edges that end at x.

Definition 5.16. For digraphs, the **out-degree** of a vertex x is the number of edges that start at x.

Definition 5.17. A graph is **connected** if $\forall x, y \in V$, $\exists$ path from x to y.

Definition 5.18. A digraph is **strongly connected** if $\forall x, y \in V$, $\exists$ paths from x to y and y to x.

Definition 5.19. A **subgraph** of $G = (V, E)$ is a graph $H = (U, F)$ where $U \subset V$, $F \subset E$, where all elements of F must involve vertices in U.

Definition 5.20. An **induced subgraph** $H = (U, F)$ of $G = (V, E)$ is a subgraph such that if $e \in E$ and $e = \{x, y\}$ where $x, y \in U$ then $e \in F$. We write $G(U)$ because an induced subgraph is determined by its vertices.

Definition 5.21. A **component** of a graph is a maximal connected subgraph.

Theorem 5.22 (Handshake Lemma)
The number of vertices having odd degree is even.

Proof. First,

$$\sum_{x \in V} \deg(x) = \sum_{e \in E} 2$$

is even. However,

$$\sum_{x \in V} \deg(x) = \sum_{\substack{x \in V \\ \deg(x) \text{ odd}}} \deg(x) + \sum_{\substack{x \in V \\ \deg(x) \text{ even}}} \deg(x).$$

Then $\sum_{\substack{x \in V \\ \deg(x) \text{ odd}}} \deg(x)$ is even. Only an even number of odd terms can sum to an even term. Hence there are an even number of vertices. □

§5.2 Graph Representations

Usually, we assume $V = \{0, 1, \ldots, v-1\}$ where we say v is the **order** of G. Furthermore, $|E|$ is the **size** of G.

The maximum number of edges in a simple graph is $\binom{|V|}{2}$.

One method that has deteriorated in popularity is the **adjacency matrix**. We condense the graph into a matrix (a_{ij}) where $a_{ij} = \begin{cases} 0 & \text{if no edge } \{i,j\} \\ 1 & \text{if edge } \{i,j\} \end{cases}$.

Definition 5.23. A **tree** is a connected acyclic graph.

Definition 5.24. A **forest** is an acyclic graph.

Definition 5.25. A **DAG** is an directed acyclic graph.

Definition 5.26. A **complete** graph is one with edges between all pairs of vertices. This is denoted by K_n (n vertices).

Definition 5.27. A cycle having n vertices is C_n.

Definition 5.28. A graph is **bipartite** if $V = V_1 \cup V_2$ (where $V_1 \cap V_2 = \emptyset$) and every edge $\{x, y\} \in E$ has $x \in V_1$, $y \in V_2$. $K_{m,n}$ has $|V_1| = m$, $|V_2| = n$, and every edge between V_1 and V_2.

An adjacency matrix is symmetric for undirected graphs, and arbitrary for digraphs. Adjacency matrices take up $\theta(n^2)$ space.

There is another representation called the **adjacency list**, which takes up $\theta(V + E)$ space.

We have notions of **sparse** and **dense** graphs. Sparse graphs have few edges, while dense graphs have lots of edges.

Adjacency lists are implemented as follows: each element in an array whose index is i is a pointer that refers to an adjacency list, which consists of all neighbors of i.

Adjacency lists are usually better than adjacency matrices, so when in doubt, use adjacency lists.

§5.3 Weighted Graphs

In a weighted graph, each edge (arc) has a weight $w : E \to \mathbb{R}$. An example of implementation:

```
struct Edge {
    Edge (int to = -1, int wt = DEFAULT_WEIGHT);
    int to;
    int wt;
};
struct Vertex {
    Vertex();
    ~Vertex();
    int id;
    vector<Edge> nbrs;
    int deg; // Redundant: nbrs.size()
};
```

There are many **intractable** questions on graphs:

Definition 5.29. A **clique** is a complete subgraph.

Consider a book of friends in BCA. Every student has a list of students in BCA who are friends. Vertices are students. Edges join 2 students who are mutual friends. What is the largest clique?

This problem is completely intractable.

Definition 5.30. A **Hamiltonian cycle** is a cycle that contains all vertices.

Definition 5.31. A **vertex cover** is a set of students who are friends with every student at BCA.

What is the minimum vertex cover?

Definition 5.32. A **dominating set** is a set of vertices such that every edge is incident on some member of the set.

What is the minimum dominating set?

§5.4 Graph Traversal

Definition 5.33. The edges that appear in a DFS tree are called **tree edges**.

Definition 5.34. The edges not in the DFS tree are called **back edges**.

Proposition 5.35

If there are no back edges then the graph is a forest.

Definition 5.36. The **pre-order number** for a vertex is assigned when first visited.

Definition 5.37. The **post-order number** is assigned upon retreat to its parent.

The number of restarts in DFS (all of current vertex numbers are visited and no parent) is equal to the number of components.

This takes $O(V + E)$ (linear complexity) time.

Types of arcs:

1. Tree
2. Back (to an ancestor)
3. Forward (to a descendant)
4. Cross (none of the above)

§5.5 Topological Sorting

Given a **partial order** on a set (poset): this is the set of ordered pairs that is anti-symmetric and transitive. Given $(a, b), (b, c) \in \text{Ord}$, $(a, c) \in \text{Ord}$.

Definition 5.38. A **topological sort** is a permutation π such that if $(a, b) \in \text{Ord}$ then the index of a is less than the index of b.

Theorem 5.39
Suppose G is a DAG and a DFS is performed on it, while numbering vertices pre and post-order. The vertices in **reverse post order** are topologically sorted.

§5.6 Strong Components

Theorem 5.40 (Kosaraju's Algorithm for Strong Components)
Given digraph G,

1. Perform DFS on G, output vertices in reverse post-order.
2. Perform DFS on G^T, which is G with all arcs reversed, choosing vertices (if there's a choice in the order from step 1).

This takes $O(V + E)$ time.

Definition 5.41. G is **2-connected** iff the removal of no single vertex disconnects G, but $\exists 2$ vertices whose removal disconnects G.

Definition 5.42. A graph that is 2-connected is called a **block**.

For example, any cycle is a block. In a block, every edge is part of a cycle.

Definition 5.43. If a vertex's removal disconnects G then it is an **articulation point** or **cut vertex**.

Definition 5.44. An edge that is part of no cycle is a **bridge**.

§5.7 Finding Paths

The easy way to solve the shortest path problem (minimum number of edges in a path from u to v) is to run a BFS (breadth-first search), which takes $O(V + E)$ time.

A well-known intractible problem is the problem of finding a path from u to v having the most number of edges.

Definition 5.45. A **Hamiltonian Path** is a path from vertex u to v that visits every vertex in the graph exactly once.

Definition 5.46. A **Hamiltonian cycle** is a cycle that visits every vertex in a graph exactly once.

Lemma 5.47
A graph is 2-colorable iff it is bipartite.

In a bipartite graph, there are no odd cycles.

Theorem 5.48
Any bipartite graph having odd order is not Hamiltonian.

Definition 5.49. An **Euler trail** or **Euler Circuit** visits every edge exactly once.

One of the famous problems in graph theory related to this is the Bridges of Königsberg.

Lemma 5.50
A graph has an Euler circuit iff every vertex has even degree.

Lemma 5.51
A graph has an Euler trail iff $\exists!2$ vertices of odd degree.

One particular example is the Chinese Postman problem. It asks: what is the shortest walk that visits every edge in a graph?

If a graph is Eulerian, then the answer is simply that Eulerian path.

If the graph is not Eulerian, then it has an even number of odd vertices.

Another example is the Weighted Bipartite Matching problem. The aim to assign N workers to N jobs in a way that the total time is minimized. Here, the time for worker w_x to do job j_y is $M[x][y]$, where M is a given matrix for this problem. To solve this, we use the Hungarian algorithm.

Definition 5.52. A **matching** is a set of pairwise, non-adjacent edges.

Definition 5.53. A **perfect matching** is one in which all vertices are matched.

Hall's Marriage Theorem provides a solution to the simple perfect matching problem.

"the progressive ones among us would be happy to know that gay matching is polynomial time" - Nevard 1/10/20

"It takes three to tango, on Mars" - Nevard 1/10/20

§6 Logic

§6.1 Propositional

Some examples of **atomic sentences** are "My name is John" and "I am 39 years old," where each is a declaration that is either true or false, and cannot be broken down further.

§6.2 The Satisfiability Problem

Definition 6.1. A **literal** is an atomic variable or its negation.

Definition 6.2. A **clause** is a **disjunction** of literals.

Definition 6.3. An expression in **Conjunctive Normal Form (CNF)** is a conjunction of clauses.

Given an expression like

$$(p_1 \vee \sim p_2 \vee p_3) \wedge (\sim p_2 \vee p_3 \vee p_5) \wedge (\sim p_4 \vee \sim p_5 \vee \sim p_6),$$

(but doesn't have to be in CNF form) for which values of $p_1, \ldots, p_6$ satisfy this expression? This problem is intractible and is famous for being the first NP-complete problem.

The 3-SAT problem is a simpler version of this problem, where each clause has exactly 3 literals. But this is still just as hard as general satisfiability.

What about 2-SAT (each clause has exactly two literals)? The good news is that this is efficiently solvable.

In fact, we can use strong components to solve 2-SAT. We turn a given expression into a directed graph. Note that $p \implies q \equiv \sim p \vee q$.

For example, given

$$(p_4 \vee \sim p_5) \wedge (\sim p_1 \vee p_2) \wedge (p_3 \vee p_5) \wedge (\sim p_2 \vee \sim p_1),$$

note that this is equivalent to

$$(\sim p_4 \implies \sim p_5) \wedge (p_1 \implies p_2) \wedge (\sim p_3 \implies p_5) \wedge (p_2 \implies \sim p_1).$$

Furthermore, note that $p \implies q \equiv \sim q \implies \sim p$. Then, we also have

$$\begin{aligned} p_5 &\implies p_2, \\ \sim p_2 &\implies \sim p_1, \\ \sim p_5 &\implies p_3, \\ p_1 &\implies \sim p_2. \end{aligned}$$

Thus, we turn each implication into an arc of a directed graph, with vertices p_1, $\sim p_1$, $\ldots p_5, \sim p_5$.

Proposition 6.4
The expression is not satisfiable if both $p \implies \sim p$ and $\sim p \implies p$ are in the same strong component.

§A Review of Probability

§A.1 Introduction

Our goal is to define what probability means intuitively. There are many different schools of thought about probability:

- "Degree of Belief": A probability represents how certain we are that an event will occur.

- "Frequentist": A probability of an event is the number of occurrences of the event divided by the number of trials.
- Bayesian

And others.

There are important people like Juden Pearl and Fisher Pearson (who literally does not exist i.e. doesn't come up on Google search) who did important things.

The modern approach is to axiomatize probability. The **Law of Large Numbers** states that if an event has a probability P, then

$$\lim_{\text{number of trials}\to\infty} \frac{\text{number of occurrences}}{\text{number of trials}} = P.$$

In order to rigorize probability, we need to introduce an important definition:

§A.2 Probability Spaces

Definition A.1. A **Probability Space** consists of a **state space**, and a number assigned to each element of the state space. For example, if our probability space corresponds to tossing a coin, then the state space is $\{H, T\}$, and each of these two outcomes is assigned $\frac{1}{2}$.

The numbers we assign, called probabilities, need to satisfy the following:

1. For any $s \in S$, $0 < \mathbb{P}(s) \leq 1$.
2. $\sum_{s\in S} \mathbb{P}(s) = 1$.

A slightly more complicated probability space is flipping a coin n times for some integer n. This has a state space $\Omega = \{(e_1, e_2, \ldots, e_n) : e_i \in \{H, T\}\}$, so that $|\Omega| = 2^n$.

We define an **event** to be a subset of Ω. For example, the event "There are exactly $\frac{n}{2}$ heads" could be characterized as the subset

$$A = \{(e_1, e_2, \ldots, e_n) : \sum_{i=1}^{n} e_i = \frac{n}{2}\}.$$

Then the cardinality of A is $\binom{n}{n/2}$.

Definition A.2. A **probability distribution** is a function $P : \Omega \to [0, 1]$ satisfying $\forall \omega \in \Omega$,

$$0 \leq P(\omega) \leq 1,$$
$$\sum_{\omega\in\Omega} P(\omega) = 1.$$

For example, on the previous example state space, a valid distribution would be giving everything a probability of 2^{-n}, i.e.

$$P(e_1, e_2, \ldots, e_n) = 2^{-n}$$

A set of events satisfying a probability distribution with probabilities summing to 1 is called **exhaustive and mutually exclusive**.

However, if we want to model a coin with a $\frac{2}{3}$ chance of flipping heads, then the probability distribution would be

$$P(e_1, e_2, \ldots, e_n) = \left(\frac{2}{3}\right)^{\text{number of heads}} \left(\frac{1}{3}\right)^{\text{number of tails}}$$

So we obtain the **binomial distribution**, where the probability of obtaining k heads in n tosses is

$$\binom{n}{k} p^k (1-p)^{n-k}$$

Another possible probability space is the **Poisson distribution**, where we toss the coin repeatedly until we get a heads. How many tosses will we do? The state space is $\Omega = \{(e_1, e_2, \ldots, e_n) : e_1 = e_2 = \cdots = e_{n-1} = T, e_n = H\}$. The probability of an n-length state is $(1-p)^{n-1}p$ (called the **geometric distribution**).

The Binomial Theorem directly confirms that the Binomial Distribution is indeed a valid probability distribution:

$$\sum_{k=0}^{n} \binom{n}{k} p^k (1-p)^{n-k} = (p + (1-p))^n = 1.$$

Furthermore, $\sum_{n=1}^{\infty} (1-p)^{n-1}p = 1$, so the geometric distribution is also a probability distribution.

§A.3 Random Variables

Definition A.3. A **random variable** X is a function $\omega \to \mathbb{R}$. We say

$$P(X = 17) \stackrel{\text{def}}{=} P(\{\omega \in \Omega : X(\omega) = 17\}.$$

For example, consider tossing a coin n times, with random variable $X =$ the number of heads. Then

$$X(k) = \binom{n}{k} p^k (1-p)^{n-k}.$$

Definition A.4. The "average" of X is equal to the **expectation** of X which is,

$$\mathbb{E}(X) = \sum_{v \in \text{Range of } X} v \cdot P(X = v).$$

For example, consider again the toss of a coin, such that

$$X = \begin{cases} 0 & \text{tails} \\ 1 & \text{heads} \end{cases}.$$

Then $\mathbb{E}(X) = p(1) + (1-p) \cdot 0 = p$.

What if we defined the random variable X differently, i.e. let $X =$ number of heads. Then

$$\begin{aligned} \mathbb{E}(X) &= \sum_{k=0}^{n} \binom{n}{k} p^k (1-p)^{n-k} k \\ &= \sum_{k=0}^{n} k \binom{n}{k} p^k (1-p)^{n-k} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{k=1}^{n} \frac{n!}{(k-1)!(n-k)!} p^k (1-p)^{n-k} \\
&= n \sum_{k=1}^{n} \frac{(n-1)!}{(k-1)!((n-1)-(k-1))!} p^k (1-p)^{n-k} \\
&= np \sum_{j=0}^{n-1} \binom{n-1}{j} p^j (1-p)^{n-1-j} \\
&= np.
\end{aligned}
$$

With a fair coin tossed n times, the average number of heads will be $\frac{n}{2}$.

There are some important expectations. Often, let $\mu = \mathbb{E}(X)$. Since a random variable is a function, we can add and multiply functions together.

Definition A.5. If X, Y are random variables, then let

$$(\alpha X + \beta Y)(\omega) \stackrel{\text{def}}{=} \alpha X(\omega) + \beta X(\omega).$$

$$XY(\omega) \stackrel{\text{def}}{=} X(\omega) Y(\omega).$$

Definition A.6. The **variance** of X is

$$\operatorname{var}(X) = \sigma^2(X) = \mathbb{E}((X-\mu)^2).$$

Here, σ is the **standard deviation** (i.e. the square root of the expression above).

Exercise A.7. Compute the standard deviation of the uniform distribution (discrete) on $[0,1] : P(X = \frac{k}{n}) = \frac{1}{n},\ k = 1, 2, \ldots, n$.

Proposition A.8

Expectation is always linear:

$$\mathbb{E}(\alpha X + \beta Y) = \alpha \mathbb{E}(X) + \beta \mathbb{E}(Y).$$

We will not cover the proof here.

Therefore, we can find an alternate expression for variance:

$$
\begin{aligned}
\operatorname{var}(X) = \sigma^2(X) &= \mathbb{E}(X^2 - 2\mu X + \mu^2) \\
&= \mathbb{E}(X^2) - 2\mu \mathbb{E}(X) + \mu^2 \\
&= \mathbb{E}(X^2) - \mu^2.
\end{aligned}
$$

From Exercise A.7, we compute

$$\sigma^2 = \sum_{k=1}^{n} \frac{1}{n} \frac{k^2}{n^2} - \mu^2 = \frac{(n+1)(2n+1)}{6n^2} - \mu^2.$$

Definition A.9. Events A, B are **independent** if $P(A \cap B) = P(A)P(B)$.

Definition A.10. Random variables X, Y are **independent** if $P(X = i, Y = j) = P(X = i)P(Y = j),\ \forall i, j$.

Proposition A.11

If X and Y are independent random variables then $\mathbb{E}(XY) = \mathbb{E}(X)\mathbb{E}(Y)$.

§A.4 Conditional Probability

Definition A.12. If B is an event such that $P(B) > 0$ then the **conditional probability** of A given B is

$$P(A \mid B) \stackrel{\text{def}}{=} \frac{P(A \cap B)}{P(B)}.$$

One thing that characterizes Bayesian probability is that this notion of conditional probability is more "fundamental" than a probability space. Virtually all real-world probability is conditional.

Fix event B such that $P(B) \neq 0$. Then define $Q(A) = \frac{P(A \cap B)}{P(B)}$. Then Q is a probability function.

Definition A.13. A and C are **conditionally independent** given B if $P((A \cap C) \mid B) = P(A \mid B)P(C \mid B)$.

We can rewrite the definition of conditional probability as

$$P(B)P(A \mid B) = P(A \cap B) = P(B \cap A) = P(B \mid A)P(A).$$

The following result immediately follows:

Theorem A.14 (Bayes' Theorem)

$$P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A)}.$$

For example, consider the model of diseases and symptoms. By Bayes' Theorem,

$$P(\text{disease} \mid \text{symptom}) = \frac{P(\text{symptom} \mid \text{disease})P(\text{disease})}{P(\text{symptom})}.$$

The values $P(\text{disease})$ and $P(\text{symptom})$ are known as **prior probabilities** while $P(\text{symptom} \mid \text{disease})$ is a **posterior probability**. It is much easier to estimate the probability that some disease will cause a symptom, rather than the other way around.

Suppose $B = \bigcup_{i=1}^{n} B_i$ where $B_i \cap B_j = \emptyset$ for $i \neq j$. Then

$$A \cap B = \bigcup_{i=1}^{n} (A \cap B_i),$$

which implies

$$P(A \cap B) = \sum_{i=1}^{n} P(A \cap B_i).$$

Then

$$P(A \mid B) = \frac{\sum_{i=1}^{n} P(A \mid B_i)P(B_i)}{\sum_{i=1}^{n} P(B_i)}.$$

In the special case that $B = \bigcup_{i=1}^{n} B_i = \Omega$, we have the significant simplification

$$P(A) = \sum_{i=1}^{n} P(A \mid B_i)P(B_i).$$

This is known as the **Law of total probability**.

Looking back to our example with disease and symptom, we can use this law to compute

$$P(\text{symptom}) = \sum_{i=1}^{n} P(\text{symptom} \mid \text{health condition}) P(\text{health condition}).$$

Then recalling that

$$P(\text{disease} \mid \text{symptom}) = \frac{P(\text{symptom} \mid \text{disease}) P(\text{disease})}{P(\text{symptom})},$$

it is difficult to compute $P(\text{disease})$. But suppose we were interested in the **odds**,

$$P(\sim \text{disease} \mid \text{symptom}) = \frac{P(\text{symptom} \mid\sim \text{disease}) P(\sim \text{disease})}{P(\text{symptom})}.$$

If we fix the symptom and let $Q(\text{disease}) = P(\text{disease} \mid \text{symptom})$, we have $Q(\text{disease} + Q(\sim \text{disease}) = 1$. Then, we can consider the ratio,

$$\frac{P(\text{disease} \mid \text{symptom})}{P(\sim \text{disease} \mid \text{symptom})} = \frac{P(\text{symptom} \mid \text{disease})}{P(\text{symptom} \mid\sim \text{disease})} \cdot \frac{P(\text{disease})}{1 - P(\text{disease})}.$$

Often, Bayes' Theorem is more useful in this form. We call $\frac{P(A|B)}{P(A|\sim B)}$ the **likelihood ratio** and $\frac{P(A)}{1-P(A)}$ the **odds** of A.

§A.5 Continuous Probability

Consider randomly breaking a stick: What is the ratio of the shorter stick to the longer stick?

This experiment has an uncountable number of outcomes. We replace $P(\omega)$ by $P(a \leq X \leq b)$, given by a **probability density**.

Definition A.15. $P(a \leq X \leq B) = \int_a^b p(x)dx$, where $p(x) \geq 0$ and $\int_\Omega p(x)dx = 1$.

Essentially, the sums turn into integrals.

If $p(x)$ is the probability density of a random variable X, then intuitively, we define

$$\mathbb{E}[X] = \int_X xp(x)dx$$

Then the variance of X is:

$$\text{Var}[X] = \int_X (x-\mu)^2 p(x)dx = \left(\int_X x^2 p(x)dx\right) - \mu^2.$$

§A.6 Common Continuous Probability Distributions

The **uniform distribution** on $[0, 1]$ has $p(x) = 1$ for all x. Then the expected value of X is

$$\int_0^1 xdx = \frac{1}{2}.$$

The variance is $\int_0^1 \left(x - \frac{1}{2}\right)^2 dx = -\frac{1}{4} + \int_0^1 x^2 = \frac{1}{12}$.

Theorem A.16 (Central Limit Theorem)
Let $\{X_n\}$ a sequence of independently and identically distributed random variables, plus some additional hypotheses that Dr. Nevard forgot. Let $z_n = \left(\frac{X_n - \mu}{\sigma}\right)$, where $\mathbb{E}[z_n] = 0$ and $\text{var}(z_n) = 1$. Then

$$\frac{z_1 + \ldots + z_n}{n}$$

converges to the **normal distribution** (also called Gaussian distribution) with mean 0 and variance 1.

The standard normal distribution on $\mathbb{R}$ is

$$p(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

It is a very nice bell curve.

Exercise A.17. Given that $\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} = 1$, compute its variance.

An example of a density with infinite variance is the **Cauchy distribution**.

$$\int_{-\infty}^{\infty} \frac{1}{1+x^2}\, dx = \tan^{-1} x \Big|_{-\infty}^{\infty} = \pi.$$

From this, $p(x) = \frac{1}{\pi} \frac{1}{1+x^2}$ is a probability density. Then

$$\mu = \frac{1}{\pi} \int_{-\infty}^{\infty} \frac{x}{1+x^2}\, dx = 0.$$

Thus, the variance is

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2}\, dx = \int_{-\infty}^{\infty} 1 - \frac{1}{1+x^2}\, dx,$$

which does not exist.

Given a probability density function $p(x)$, we define the **cumulative density function** (CDF) of the distribution to be

$$F(x) = \int_{-\infty}^{x} p(t) dt$$

For example, if $p(x)$ is the normal distribution with mean 0 and variance 1, then

$$F(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-t^2/2}\, dt$$

The CDF is almost always continuous and nondecreasing, by the nature of probability distributions.

Problem A.18. Given `bool bad_flip()` which returns `true` with probability p and `false` with probability $1 - p$, where p is constant but unknown, then write `bool good_flip()` which returns `true` with probability $\frac{1}{2}$ and `false` with probability $\frac{1}{2}$, using `bad_flip` as a source of stochasticity.

Solution. First, run `bad_flip()` two times to get two flipped coins. If they are the same, then run `bad_flip()` until you get two different coins. Given that they are different, the probability that the first coin is heads is $\frac{1}{2}$, and vice-versa. So, just return the first coin.

To see why, consider conditional probability:

$$P(\text{first flip heads} \mid \text{both flips different}) = \frac{p(1-p)}{p(1-p)+(1-p)p} = \frac{1}{2}. \qquad \square$$

Problem A.19. Given `bool good_flip()` which returns `true` with probability $\frac{1}{2}$, write `bool prob(long a, long b)` which returns `true` with probability $\frac{a}{b}$ exactly (not an approximation). This should require $O(1)$ flips.