```
====================
        STACKS
==================== //all interfaces allow these interactions with stacks:
    create an empty stack
    push an item onto the stack //items are always added to and removed from the TOP of the stack
    pop an item off the stack
    look at the top item of the stack
    is the stack empty?
------ //some interfaces let you do these:
    how many items are on the stack?
    look at any item on the stack

#include <stack>
using namespace std;

int main()
{
    stack<int> s;
    s.push(10);
    s.push(20);
    if (!s.empty())
        cout << s.size(); //size is 2
    s.pop(); //pop an item off the top
    int n = s.top(); //n is the top item on the stack, which is 10 here

}

//Maze function using stacks

class Coord
{
public:
    Coord(int rr, int cc) : m_r(rr), m_c(cc) {}
    int r() const { return m_r; }
    int c() const { return m_c; }
private:
    int m_r;
    int m_c;
};

bool pathExists(char maze[][10], int sr, int sc, int er, int ec)
{
    stack<Coord> coordStack;
    Coord start(sr, sc);
    Coord end(er, ec);

    coordStack.push(start);
    maze[start.r()][start.c()] = '#';

    while (!coordStack.empty())
    {
        Coord loc = coordStack.top();
        coordStack.pop();

        if ((loc.r() == end.r()) && (loc.c() == end.c()))
            return true;

        if (maze[loc.r() + SOUTH][loc.c()] == '.') //code puts all empty spaces around location onto the stack, and eventually checks them all, resulting in every accessible maze location being checked
        {
            Coord newloc(loc.r() + SOUTH, loc.c());
            maze[newloc.r()][newloc.c()] = '#';
            coordStack.push(newloc);
        }
        if (maze[loc.r()][loc.c() + WEST] == '.')
        {
```

```
            Coord newloc(loc.r(), loc.c() + WEST);
            maze[newloc.r()][newloc.c()] = '#';
            coordStack.push(newloc);
        }
        if (maze[loc.r() + NORTH][loc.c()] == '.')
        {
            Coord newloc(loc.r() + NORTH, loc.c());
            maze[newloc.r()][newloc.c()] = '#';
            coordStack.push(newloc);
        }
        if (maze[loc.r()][loc.c() + EAST] == '.')
        {
            Coord newloc(loc.r(), loc.c() + EAST);
            maze[newloc.r()][newloc.c()] = '#';
            coordStack.push(newloc);
        }
    }
    return false;
}

==========================
QUEUES //like a stack but backwards
=========== //all interfaces allow these interactions with queues
    create an empty queue
    enqueue an item onto the queue //items are always added to and removed from the BACK of the queue
    dequeue an item from the queue
    look at the front item of the queue
    is the queue empty?
    ------ //some interfaces let you do these
    how many items are in the queue?
    look at the back item of the queue
    look at any item in the queue


have to #include <queue>

//Maze function using queues
bool pathExists(char maze[][10], int sr, int sc, int er, int ec)
{
    queue<Coord> coordQueue;
    Coord start(sr, sc);
    Coord end(er, ec);

    coordQueue.push(start);
    maze[start.r()][start.c()] = '#';

    while (!coordQueue.empty())
    {
        Coord loc = coordQueue.front();
        coordQueue.pop();

        if ((loc.r() == end.r()) && (loc.c() == end.c()))
            return true;

        if (maze[loc.r() + SOUTH][loc.c()] == '.') //code puts all empty spaces around location onto the stack, and eventually checks them all, resulting in every accessible maze location being checked
        {
            Coord newloc(loc.r() + SOUTH, loc.c());
            maze[newloc.r()][newloc.c()] = '#';
            coordQueue.push(newloc);
        }
        if (maze[loc.r()][loc.c() + WEST] == '.')
        {
            Coord newloc(loc.r(), loc.c() + WEST);
            maze[newloc.r()][newloc.c()] = '#';
            coordQueue.push(newloc);
        }
```

```
        if (maze[loc.r() + NORTH][loc.c()] == '.')
        {
            Coord newloc(loc.r() + NORTH, loc.c());
            maze[newloc.r()][newloc.c()] = '#';
            coordQueue.push(newloc);
        }
        if (maze[loc.r()][loc.c() + EAST] == '.')
        {
            Coord newloc(loc.r(), loc.c() + EAST);
            maze[newloc.r()][newloc.c()] = '#';
            coordQueue.push(newloc);
        }
    }
    return false;
}
```

====================
Notations
====================

```
prefix notation:
f(x,y,z)
add(sub(8, div(6,2), 1) //also works: add sub 8 div 6 2 1       also works: + - 8 / 6 2 1

infix notation:
8-6/2+1  //the same thing as the line above, can be more confusing for a human to parse sometimes?

postfix notation:
8 6 2 / - 1 +   //once again, the same thing
((8 (6 2 /) -) 1 +) //the associated groupings.
//Postfix notation doesn't need any additional specificity. To the computer, it's always unambiguous what it operates on
//postfix is actually easier to process than prefix/infix notation, runs faster than both. Common expression evaluation: given something in infix, translate it to postfix, then it's easy to evaluate

8 6 2 / - 1 +
//  evaluating a postfix sequence: (pseudocode)

    operand stack
    run through the postfix expression:
        when you encounter a number: push it onto the stack
        when you encounter an operand: pop off the top two numbers, run the expression, push the result back onto the stack
    if this is a valid postfix expression, the stack will have exactly one value on it (the value of the expression)


//  translating infix to postfix:

    make an operator stack and a postfix string

    run through the expression:
        numbers: push to the postfix string
        operator:
            open parens ->always push
            top of stack is parens -> always push
            current operator has a higher precedence than top of stack: push
            close parens: pop the stack to the postfix string until you pop an open parens
    if the current operator is lower precedence than what's on top of the stack, it goes lower down
    //try it with some numbers! - see if it works
```

==========================
Making a picture-drawing algorithm

```
class Shape //generalization of the idea of circles and rectangles
```

```
{
    void move(double xnew, double ynew);
    virtual void draw() const;   //draw is called differently for each type of shape.
    double m_x;
    double m_y;
};

class Circle : public Shape //introducing a new type called circle, and it's a kind of shape
{
    virtual void draw() const;
    double m_r; //since circle is a type of shape, you only have to declare member functions that are unique to it. it already has move, m_x, m_y, etc. Draw seems to be a special case watch out.
};

class Rectangle : public Shape
{
    virtual void draw() const;
    double m_dx;
    double m_dy;
};

Shape* pic[100];  //Shape is a heterogeneous collection of items in a strongly-typed language
pic[0] = new Circle; //circle "is-a-kind-of" // "is-a" shape
pic[1] = new Rectangle;
pic[2] = new Circle;
//data abstraction is pretty powerful


for (int k = 0; k < ...; k++)
    pic[k]->draw();

void f(Shape& s)
{
    s.move(10, 20);
    s.draw();
}

void Shape::move(double xnew, double ynew)
{
    m_x = xnew;
    m_y = ynew;
}

void Shape::draw() const
{
    ...draw a cloud centered at m_x, m_y...
}

void Circle::draw() const
{
    ...draw a circle...
}

void Rectangle::draw() const
{
    ...draw a rectangle...
}
```

==
review sesh notes

member variables are constructed before the constructor is called -> inside out
member variables are destructed after the desctructor is called -> outside in

if member variables are dynamically allocated, they will not be deleted automatically when/after the destructor is called - gotta add that to the destructor yourself to get it to happen

default copy constructor just copies member variables from the existing object to the new one

you need your own copy constructor when:

    the object contains a pointer to something in memory. otherwise, the pointers will point to the same block of memory, instead of just being a copy. This means that when you modify one the other changes too.
    (^ dynamic memory allocation)

=====================================
Inheritance / Virtual Functions
=====================================

```
//if a class is designed to be a base class, declare a virtual destructor and implement it
// STEPS OF CONSTRUCTING OBJECTS: THE FIRST STEP
//  Construct the Base Part
//  Construct the Data Members
//  Execute the body of the constructor

 STEPS OF DESTRUCTION
    1. Execute the body of the destructor
    2. Destroy the data members
    3. Destroy the base part //new third step

class Device //device is the base class for several other things.
{
    public:
    virtual ~Device(){}
    virtual void open() = 0;
    virtual void write(char c) = 0;
    virtual void close() = 0;
};

class BannerDisplay : public Device
{
public:
    virtual void open(); //implemented somewhere
    virtual void write(char c); //implemented somewhere
    virtual void close(); //implemented somewhere
private:
    ...
};

class Modem : public Device
{
public:
    virtual void open(); //implemented somewhere
    virtual void write(char c); //implemented somewhere
    virtual void close(); //implemented somewhere
private:
    ...
};

void writeString(Device& d, string msg)
{
    for (int k = 0; k != msg.size(); k++)
        d.write(msg[k]);
}


//program to send warning messages in various mediums
enum CallType {
    VOICE, TEXT
};
```

```
class Medium
{
public:
    Medium(string id) : m_id(id) {}
    virtual ~Medium() {}
    virtual string connect() const = 0;
    virtual string transmit(string msg) const { return "text: " + msg; }
    string id() const { return m_id; }
private:
    string m_id;
};

class Phone : public Medium
{
public:
    Phone(string num, CallType type) : Medium(num) { m_type = type; }
    ~Phone() { cout << "Destroying the phone " << id() << "." << endl; }
    string connect() const { return "Call"; }
    string transmit(string msg)  const
    {
        switch (m_type)
        {
        case VOICE:
            return "voice: " + msg;
        case TEXT:
            return "text: " + msg;
        }
    }
private:
    CallType m_type;
};

class TwitterAccount : public Medium
{
public:
    TwitterAccount(string id) : Medium(id) {}
    ~TwitterAccount() { cout << "Destroying the Twitter account " << id() << "." << endl; }
    string connect() const { return "Tweet"; }
};

class EmailAccount : public Medium
{
public:
    EmailAccount(string id) : Medium(id) {}
    ~EmailAccount() { cout << "Destroying the email account " << id() << "." << endl; }
    string connect() const { return "Email"; }
};

void send(const Medium* m, string msg)
{
    cout << m->connect() << " using id " << m->id()
        << ", sending " << m->transmit(msg) << endl;
}
```

================================
        RECURSION
================================

I want to sort a pile of N items:

```
    if (N > 1)
    {
        split the pile about evenly into two insorted piles
        sort the left subpile
        sort the right subpile
        merge the two sorted subpiles into one sorted pile
```

```
    }
//to debug, assume a recursive function works
    void sort(int a[], int b, int e)
    {
        if (e - b >= 2)
        {
            int mid = (b + e) / 2;
            sort(a, b, mid);
            sort(a, mid, e);
            merge(a, b, mid, e);
        }
    }

    int main()
    {
        int arr[5] = {40, 30, 20, 50, 10};
        sort(arr, 0, 5);
    }




To prove P(N) for all N >= 0:
    1. Prove: P(0)
    2. Prove: If P(k) is true for all k < N, then P(N)


//Some recursive functions

bool somePredicate(string s) //predicate to check if the string has things in it. Returns true if the string is NOT empty, and false otherwise
{
    return !s.empty();
}

bool allTrue(const string a[], int n) // Return false if the somePredicate function returns false for at least one of the array elements; return true otherwise.
{
    if (!somePredicate(a[n - 1])) //if the item fails the predicate, return false
        return false;

    if ((n - 1) > 0) //if it doesn't fail and it's not the last item in the list, check the next one
        if (allTrue(a, n - 1))
            return true; //the return must be conditional on allTrue, not just after it, or the function returns true every time
}

int countFalse(const string a[], int n) // Return the number of elements in the array for which the somePredicate function returns false.
{
    int i = 0;
    if (n > 0)
    {
        if (!somePredicate(a[n - 1]))
            i += (1 + countFalse(a, n - 1));
        else
            i += (0 + countFalse(a, n - 1));
    }
    return i;
}

// Return the subscript of the first element in the array for which
// the somePredicate function returns false.  If there is no such
// element, return -1.
int firstFalse(const string a[], int n)
{
    if (n <= 0)
        return -1;
```

```
    int x = firstFalse(a, n - 1);

    if (!somePredicate(a[n - 1]) && x == -1)
        return n - 1;
    return x;
}

// Return the subscript of the least string in the array (i.e.,
// return the smallest subscript m such that a[m] <= a[k] for all
// k from 0 to n-1).  If the function is told to examine no
// elements, return -1.
int indexOfLeast(const string a[], int n)
{
    if (n <= 0)
        return -1;
    int i = 0;
    if (n - 1 > 0)
    {
        int comp = indexOfLeast(a, n - 1);
        if (a[comp] < a[n - 1])
            i = comp;
        else
            i = n - 1;
    }
    return i;
}

// If all n2 elements of a2 appear in the n1 element array a1, in
// the same order (though not necessarily consecutively), then
// return true; otherwise (i.e., if the array a1 does not include
// a2 as a not-necessarily-contiguous subsequence), return false.
// (Of course, if a2 is empty (i.e., n2 is 0), return true.)

bool includes(const string a1[], int n1, const string a2[], int n2)
{
    bool i = false;
    if (n1 > 0 && n2 > 0)
    {
        if (a1[0] == a2[0])
        {
            a2++;
            n2--;
        }
        a1++;
        n1--;
        i = includes(a1, n1, a2, n2);
    }
    if (n2 == 0)
        return true;
    else
        return i;
}
```

```
======================
trees?
======================
```

it's basically just applied math, very little actual programming

N	2N^2+1000N+1000	3N^2-2N+5	10000N
100	123,000	29,805	1,000,000
1000	3,001,000	2,998,005	10,000,000
10000	210,001,000	299,980,005	100,000,000

```
100000          20,100,001,000            29,999,800,005          1,000,000,000

Exponents totally dominate other terms when N is large (duh)


A function f(N) is O(g(N)) if there exist N0 and k such that for all N >= N0, |f(N)| <= k*g(N)

f(N) is "order g(N)"



for (int i = 0; i < N; i++) <-------------------- O(N)
    c[i] = a[i] * b[i]; <--------------- O(1)

for (int i = 0; i < N; i++) <------------------------------------------------- O(N^2) (since there's 2 O(N) in there)
{ <----------------------------------------------- O(N)
    a[i] *= 2;   <---------------------- O(1)
    for (int j = 0; j < N; j++)  <--------------- O(N)
        d[i][j] = a[i] * c[j]; <-------- O(1)
}

for (int i = 0; i < N; i++) <------------------------------------------------- O(N^2)
{ <--------------------------------------------- O(i)
    a[i] *= 2; <------------------------ O(1)
    for (int j = 0; j < i; j++) <-------------- O(i)
        d[i][j] = a[i] * c[j]; <-------- O(1)
}

for (int i = 0; i < N; i++) <------------------------------------------------ O(N^2)
{
    if (std::find(a, a+N, 10*i) != a+N) <------- Condition: O(N) //how much time do if statements account for?
        count++; <---------------------- O(1)
}

for (int i = 0; i < N; i++) <---------------------------------------------------------- O(N^2 log N)
{
    a[i] *= 2; <------------------------------------ O(1)
    for (int j = 0; j < N; j++) <--------------------------------- O(N log N)
        a[i][j] = f(a, N); //suppose f(a, N) is O(log N) <------ O(log N)
}

for (int i = 0; i < R; i++) <----------------------------------------- O(Rc log c)
{
    a[i] *= 2; <------------------------------------ O(1)
    for (int j = 0; j < c; j++) <---------------------- O(c log c) //Insertion Sort can be based off a constant runtime - 'order N'
        ... f(...c...)...; //this is o(log c)
}


there are actually tons of different sorting algorithms - the most efficient one is determined by the type and ordering of data
//compare an item to the ones before it, to figure out where it belongs. move accordingly
Selection Sort, Insertion Sort, Bubble Sort
insertion sort has a circumstance in which it's order N
Merge Sort - efficient, but needs additional space
quicksort is the fastest on average...


Quicksort is N log N!

// find a pivot, move everything smaller to the left and everything larger to the right
// do this recursively

//Best case: T(N) = O(N) + 2T(N/2) (time to pick a pivot is O(N), time to sort the
```

```
halves is T(N/2)
{ _ _ _ _ _ _ _ _ _ _ }
 [-------] [-------]
 [--] [--] [--] [--]
 [][] [][] [][] [][]

//Worst case: O(N) + T(N - 1) + T(0)
{ _ _ _ _ _ _ _ _ _ _ }
 [-------------] [-]


 //ACTUAL

 //QUICKSORT

 //ALGORITHM
 //-------------------------------------
 //swaps stuff
 void exchange(string& x, string& y)
{
    string t = x;
    x = y;
    y = t;
}

// Rearrange the elements of the array so that all the elements
// whose value is < separator come before all the other elements,
// and all the elements whose value is > separator come after all
// the other elements.  Upon return, firstNotLess is set to the
// index of the first element in the rearranged array that is
// >= separator, or n if there is no such element, and firstGreater is
// set to the index of the first element that is > separator, or n
// if there is no such element.
void separate(string a[], int n, string separator,
    int& firstNotLess, int& firstGreater)
{
    if (n < 0)
        n = 0;

    firstNotLess = 0;
    firstGreater = n;
    int firstUnknown = 0;
    while (firstUnknown < firstGreater)
    {
        if (a[firstUnknown] > separator)
        {
            firstGreater--;
            exchange(a[firstUnknown], a[firstGreater]);
        }
        else
        {
            if (a[firstUnknown] < separator)
            {
                exchange(a[firstNotLess], a[firstUnknown]);
                firstNotLess++;
            }
            firstUnknown++;
        }
    }
}

// Rearrange the elements of the array so that
// a[0] <= a[1] <= a[2] <= ... <= a[n-2] <= a[n-1]
// If n <= 1, do nothing.

//the meat of the function
void order(string a[], int n)
{
    if (n <= 1)
```

```
        return;
    int midpoint = n / 2, firstgreater = n, firstnotless = 0;
    separate(a, n, a[midpoint], firstnotless, firstgreater);

    order(a, firstnotless);
    order(a + firstnotless + 1, n - firstgreater);
}



 ----------------------------------------------
 -  At 0.25 Microseconds per step (4M Steps/sec)
 -----------------------------------------------
      N          0.5 N^2      7.527 N log(2)(N)
 ========== =============== =================
        100           1.25ms            1.25ms
       1000          .125sec           1/53sec
      10000          12.5sec            1/4sec
     100000            21min          3.125sec
    1000000            35hrs           37.5sec



struct Node
{
    std::string data;
    vector<Node*> children;
}

               George
               / | \
              /  |  \
         eliza  bet  anne
          / \       / | \
         /   \     /  |  \
       wall  nut peg mar  owen


int countTree(const Node* t) //function to count the number of nodes in a tree
{
    if (t == nullptr)
        return 0;
    int total = 1;
    for (int k = 0; k != t->children.size(); k++)
        total += countTree(t->children[k])
    return total;

}

void printTree(const Node* t) //function to print a tree
{
    if (t != nullptr)
    {
        cout << t->data << endl;
        for (int k = 0; k != t->children.size(); k++)
            printTree(t->children[k]);
    }
}

int fixNegatives (Node* p) //changes any negative values to 0 in a 4-tree
{
    if (p == nullptr)
        return 0;

    int Count = 0;
    if (p->data < 0)
    {
        p->data = 0;
        Count++;
```

```
    }
    for (int i = 0; i < 4; i++)
        Count += fixNegatives(p->child[i]);
}

int countIncludes(const string a1[], int n1, const string a2[], int n2) //counts the number of times a thing is included
{
    int numofInc = 0;
    if (n1 <= 0)
    {
        if (n2 == 0)
            numofInc++;
        return numofInc;
    }

    if (a1[0] == a2[0])
        numofInc += countIncludes(a1 + 1, n1 - 1, a2 + 1, n2 - 1);

    numofInc += countIncludes(a1 + 1, n1 - 1, a2, n2);

    return numofInc;
}
```