8/29/16

Example:
MaxElements (A, n)

- Input:
    - array of integers A
    - size of array n
- Output:
    - Maximum element in A

```
    CurrentMax  A[0]
    for(i = 1; i < n-1; i++)
     do:
       if(A[i] > CurrentMax) then:
         CurrentMax  A[i]
       return CurrentMax
```

Primitive types is Java:

- Byte
- Short
- Int
- Long
- Float
- Double
- Boolean
- Char

Default values for primitive types

- Byte : 0
- Short : 0
- Int : 0
- Long : 0
- Float : 0
- Double : 0
- Char : \u0000
- String : null
- Boolean : false

We reference static code by the class name and then the name of the method. We reference Instance code with a state and then the name of the method.

9/1/16

Continuing Java

Java byte code - analogous to machine code. Encoding that Java Virtual Machine understands. You can compile on any system and it will work on any other computer because of JVM layer between the compiler and the code.

Data Extraction

This is where we create new classes and methods. OOP stuff...

Encapsulation: Hiding implementation details behind the programming language. We don't care about how the code is written, we just care about how it works, it should work when we call the method and may give it our data.

Google: Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

If we want to create a class called XYPoint:

- We have a constructor that takes two input parameters
- Two getter methods to retrieve from the object

```
XYPoint(int x, int y){
  int xPoint();
  int yPoint();
}
```

To call the object in java we would:

```
XYPoint xyp = new XYPoint(0 , 0);
```

Java would also initialize values if you dont specify, for e.g.:

```
XYPoint xpy2 = new XYPoint();
print xyp2.x
```

This will print 0 as the value is initialized unlike C.

In Java you dont have to de-allocate memory like C as it has a garbage collector.

In C we would do the same as

```
struct{
  int x;
  int y;
} * XYPoint;
```

In C we would have to use `malloc` to allocate memory to create a new object.

In Java if we have an array:

```
int a[];
for (int i =0; i < 10; i++){
  a[i] = 0;
}
```

This will throw a RunTimeException error as the array size is not declared. Should declare the array as `arrayRefVar = new dataType[arraySize];`.

Power of Encapsulation

API for `public class Date`:

```
public class Date{
Date(int mo, int day, int year)
int month(); // getter method
int day(); // getter method
int year(); // getter method
String toString(); // Overwrite default toString() method
                  //and provide custom implementation
}
```

Example implementation of the API:

```
public class Date{
  private final int month;
  private final int year;
  private final int day;

  public Date(int m, int d, int y){
    month = m;
    day = d;
    year = y;
  }

  int month(){
    return month;
  }

  int day(){
    return day;
  }

  int year(){
    return year;
  }

  String toString(){
    return month() + "\" + day() + "\" + year() + "\";
  }
}
```

If you have another class, you cannot access these `private` variables. `Final` is when we set it and never change it.

If we want to use just one integer to represent the date we have to create an encoding:

```
public class Date{
  private final int _date;

  /*
  We dont need 32 bits to represent number 0-31 for date
  Therefore we are shifting the year and month to fit all the date
  variables in just one _date variable
  */
  public Date(int m, int d, int y){
    _date = y * 512 + m * 32 + d;
  }

  int month(){
    return (_date/32)%16;
  }

  int day(){
    return _date%32;
  }

  int year(){
    return _date/512;
  }

  String toString(){
    return month() + "\" + day() + "\" + year() + "\";
  }
}
```

Mutable vs Immutable

Some data types are by nature mutable. For example string, if you alter a string a new string object is created. For example if you two strings:
`String a = "tiny"` and `String b = "cat"` and concat them using `a = a.concat(b)`. There are three string objects created here, at `a`, `b` and the last one when we concat it. Arrays are also mutable.

Mutable data types uses more memory as it forces to create a new object with every edit. We have to care about it and while writing code, if we dont want something from outside don't change our encapsulated value, we have to care about and make sure it doesn't happen. Advantage is that these data types are flexible and their values can easily be changed.

```
Public class Vector{

  private final double [] values;

  Public Vector(double [] myvals){
    value = myvals;
  }

}
```

As we said `final` it cannot be altered. Let's consider if we have the following code:

```
double [] ds = {3.0 ,4.0};
vector V  = new Vector(ds);
ds[0] = 0.0;
ds[1] = 0.0;
```

The vector may have the new values. So we intended to make our vector Immutable by saying `final`, but we just changed its values here to `0.0`. `final` says it will not change the value to where the array points, so it will point to the ds memory space. When we create our vector V, its values points to the same place as ds values.

To fix it, we create a new array inside the instance and copy it. Known as `defensive-copy` and therefore we will change our code to:

```
Public class Vector{

  private final double [] values;

  Public Vector(double [] myvals){
    value = new double [myvals.length()];
    for (int i = 0; i < myvals.length(); i++){
      value[i] = myvals[i];
    }
  }

}
```

As the size of your data grows the execution time grows linearly, as you will need a copy of the array (in our case).

Implementation of inheritance

Java provides two types of implementation inheritance:

- Sub classing
    - The notion of object inheritance. If you have parent object for e.g. `animal` and a sub-class `cat`. It inheritance all the properties of `animal` but it will also have its own properties.
    - In Java we use extends keyword to tell a class inherits from its superclass.
- Sub typing

  - Its an alternative to inheritance in Java. Instead of inherit the properties and methods of a parent class, it allows us to inherit a relationship.
  - One can take unrelated classes and define a relationship.
  - For example if we have `animals` and `spaceships`. And we want to say they both require fuel, we can define a method consume which will be an `interface` in Java.
  - We use the `implements` keyword to sub type from a different class.

Inheriting from a base class

```
1. class getClass(); - This will return an object of the class itself.
2. boolean equals(object); - Receives an object and checks equality
3. String toString(); - Prints out JavaEncoded string for the name if the
datatype provide is not string.
4. int hashCode(); - returns an integer, used for hashing
```

9/7/16

Subclassing is casting from a parent class
All classes inherit from the object class
Subtyping sets up a relationship between 2 objects
Intefaces are used for objects to inheret those functions and they are automatically part of the interface since they have the matching functions

```
interface summable {
    double sum();
}
```

for example, `vector implements summable`

if we have an object called studentGrades, it can implement summable `studentGrades implements summable`

```
class1: IntegerVector
class2: DoubleVector
```

both of these classes will implement the interface summable

The four key methods that everyone inherits from objects:

- getClass();
- toString();
- equals();
- hashCode();

Equals asks if they point to the same object, if they have the same value but are not the same object then they will not match

`hashCode();` will return an integer representation of that object AKA a fingerprint

```
Date d1 = new Date(2,4,1900);
Date d2 = new Date(4,11,2013);
Date d3 = new Date(2,4,1900);
    if(d1==d2){// this will be false
         print("EQUAL");
     }
    if(d1==d3){// this will be false
        print("EQUAL");
    }
    if(d1.equals(d3)){// this will be false
        print("EQUALS");
    }
    d3 = d1;
    if(d1.equals(d3)){// this will be true
        print("EQUALS");
    }
```

The equals method by default is the same as the double =='s.
The double equals with check if they are equal by reference AKA if they are pointing to the same memory location

But what if we want to check equality by reference?

`Override` is used to designate that you are taking the default implementation, instead of executing that run this code instead.

- if you pass `x.equals(x)` then itt must be true for all cases
- if `x.equals(y)` then `y.equals(x)` must be true
- if `x.equals(y)` and `x.equals(z)` then `z.equals(y)`

If you override equals incorrectly bad shit will happen. We need to define what will happen if the user passes null to the overridden function.

1. Null constraint, says that it should return false if null is passed
2. The overridden function must be consistent

An example below shows how to override a method

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (obj instanceof Cat) {
        Cat that = (Cat)obj;
        return (this._meows == that._meows && super.equals(that));
    }
    return false;
}
```

instead of using the `instanceof` method which breaks our code we can use the `getClass()` method to fix this

```
    if (this.getClass().equals(obj.getClass())){
        return false;
    }
```

Listed below in an example from class

```
Cat cat = new Cat("tinycat");
Animal animal = new Animal("tinycat");
system.out.println(cat.equals(animal)); // should print false but prints true
because cat is of type Animal
system.out.println(animal.equals(cat)); // should print false but prints false
```

Below is a class of IntegerSet which is an immutable type which is necessary since it will implement binary search.

```
public class IntegerSet
IntegerSet(int [])
int size()
boolean contains(int k)
```

Binary Search

Binary search checks if it is in the middle of the data set and if it is, it will return that position. If the position is greater than the key it will throw away the other half of the search and it will take the middle of that new set and repeat that process until it finds the middle of that array

---

Generics

generics let us implement a method for any type of data

The issue with the code below is that you are stuck with ints which limit the amount of information you can store.

```
public class Vector
public vector(int []);
int valueAt(int);
int size();
```

Listed below is the same class but as a generic which lets us pass any type of data in.

```
public class Vector<E>
public vector(E []);
E valueAt(int);
int size();
```

Listed below is an exmaple of using the generic class above

```
Date [] dates
Vector <Integer>
Vector <Date> vec = new Vector<Date>(Dates);
ivec = new Vector <integer>

```

9/12/16

```
a[]
b[]
int[]t = a;
a = b;
b = t;
```

This swaps the code in an efficient way. It could instead try to swap the arrays element by element but that would be a lot less efficient. This is a linear time operation.

```
Vector[] victor = new Vector[2];
Stdout.println(Victor[0].sum() + " " + victor[1].sum());
```

This will print out a null pointer exception since the object vector hasnt been created IE it has nothing to add.

```
Vector<E>
KeyValueSet<K,V,E>;
```

This says that the values K,V,E must be objects. At best they will be objects, When it is compiled it must be a type object but when it is ran you can pass whatever you want to it.

```
Public Vector<E>
    Vector(E[]);
    int size();
    E valueAt(int);
```

Class Variables:

- private final E [] _data;

Constants:

```
Public (E[] a){
    _data = (E[]) new Object[a.length];
    for(i = 0; i < a.length; i++){
        _data[i]=a[i];
        \_data[i]=a[i];
    }
}
```

methods:

```
public int size(){
    return \_data.length
}
```

```
Public E valueAt(int i){
    \_rangeCheck(i);
    return \_data[i];
}
```

```
Private void \_range_check(int i){
    if(i<0 || i >= size())
        throw new IndexOutOfBoundException("foo");
}
```

`Iterable` <E> provides one method, which returns another interface which is the `interator`<E> which has a method in it called `iterator();`.
The iterator interface has 3 key methods:
- boolean hasNext();
- E next();
- void remove(); *Will remove the last element*

InnerClass

In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java. The class written within is called the nested class, and the class that holds the inner class is called the outer class.

`private class Vector Iterator implements Iterator`<E> is an example of a of extending an interface

---

a bag is a data type in Java where you put data in and you can't take any out. It also has no iteration format.

```
Public class Bag<E> implements Iterable<E>
Bag();
void Add(E);
int size();
Iterator<E> iterator();
```

Above is an example of a bag. If you loop through the data you will go through a random path everytime.

```
Public class queue<E> implements Iterable<E>{
    Queue();
    enqueue();
    E dequeue();
    boolean isEmpty();
    int size();
}
```

This is an example of a Generic queue

```
Public class Stack<E> implements Iterable<E>{
    public Stack();
    void push();
    E pop();
    boolean isEmpty();
    int size();
}
```

9/14/16

Algorithm's to solve equations like this: (1+(2-3))

Dijkstra's two stack algorithm

Parses tokens. Tokens are the pieces of information that are useful to you
1. Create two stacks
2. Read from left to right
3. As we read, if we find an operand we will push it onto the operand stack
4. if we read an operator we will push it onto the operator stack
5. Ignore left parenthesis
6. reading right parenthesis
7. pop na operator and however many number of operands it requires
8. perform an operation
9. push the result onto the operand stack

pseudo-code

```
Operators <-- new stacks
Operands <-- new stack
.
.
.
next_item <-- read_item()
if next item is operand then:
    stack_push(operands, next_item)
```

Example
(1 + ((2 + 3)) * (4 * 5))
1. 2 + 3 = 5
2. 4 * 5 = 20
3. 20 * 5 = 100
4. 100 + 1 = 101

Linked List:

```
Private class Node{
    E data;
    Node next;
}
```

```java
import java.util.Iterator;

public class Stack<E> implements Iterable<E> {

  private Node first;   // top of the stack
  private int N;        // # of items

  private class Node {
    E item;
    Node next;
  }

  public boolean isEmpty() { return first == null; }
  public int size() { return N; }

  public void push(E item) {

    Node oldfirst = first;
    first = new Node();
    first.item = item;
    first.next = oldfirst;
    N++;
  }

  public E pop() {
    E item = first.item;
    first = first.next;
    N--;
    return item;
  }

  public Iterator<E> iterator() { return new ListIterator(); }

  private class ListIterator implements Iterator<E> {
    private Node current = first;

    public boolean hasNext() { return current != null; }

    public E next() {
      E item = current.item;
      current = current.next;
      return item;
    }

    public void remove() { /* do nothing */ }

  }
}
```

Suppose we want to implement a new constructor that takes a stack as an arg Public Stack(Stack<E> s)

We add a constructor to the Node object

```
public Node (Node e){
    item = e.item;
    next = new node(e.next);
    }
```

Order of growth classification

```
1. Constant ('c', '1') - Program where running time is independent of the size
of operators we are running it on. Example `a + b` -> constant time
2. Logarithmic - Running time is based on the size of input is expressed by `y
= c * log(N)`, where `N` represents the input size. The log is base 2 (x^A = B
--> Logx(B) = A). `Binary Search` in the IntegerSet homework is in O(log(N))
time.
3. Linear - Order of growth is directly proportional to the input size. y = c *
N. Classic single For Loop. Examples of this are copying an array by element,
finding the max in an array or printing each character in a string
4. Linearithmic - `y = c * N * log(N)` worse than linear but not as bad as
quadratic.
5. Quadratic - `y = c * N^2`, an example of this would be checking for all
pairs
6. Cubic - `y = c * N^3`, do your best to stay away from this. You do not want
to deal with this unless your input is very small
7. Exponential - `y = c * N^N` an example of this would be exhaustive search.
Check all subsets.
```

Def: Let f(n) and g(n) be functions defined on a set of natural numbers. A function is O(g(n)) if ∃ 2 positive constants c and n0 so that f(n) <= c * g(n). *g(n) is the growth function.*

Prove: `T(n) = 5n + 20` is linear O(n) by getting to f(n) <= c * g(n)
1. let n0 = 1
2. T(n)/O(n) = (5n + 20)/n <= c for all n > 1
3. observe `1 < n < n^2`
4. (5n + 20n) / n <= c
5. let c = 25
6. 5n + 20 <= 25n which is f(n) <= c * g(n)
7. 20 <= 20n and n > 1 therefore this is O(n)

9/19/16

Suppose we have a function T(N) that is O(N). Is my function also O(N^2)? Technically yes it is since there is a possibility that it could O(N^2) at some point.

Suppose we have a `T(N) = 8n^2 + 10n + 25`

Proof

We have 2 functions, g(n) and f(n) over natural numbers. We say f(n) is O(g(n)) if there exists 2 positive constants c and n0 such that f(n) is less than or equal to c * g(n) for all n greater than n0.

```
1. Let n0 = 1
2. We intend to show that T(n)/G(n) <= c for all n > 1
3. (8n^2+10n+25)/(n^2) <= c
4. Observe 1 < n < n^2 ...
5. we can then say that (8n^2 + 10n^2 + 25n^2)/(n^2) is going to meet T(n)/G(n)
<= (8n^2 + 10n^2 + 25n^2)/(n^2) = 43
6. let c = 43 and n0 = 1
7. 8n^2 + 10n + 25 <= 43n^2
8. 10n + 25 < = 35n^2
9. We can observe again that 1 < n < n^2 so that we now know that 10n + 25 will
always be less than 35n^2.
```

Unfortunately we might not know the T(n) for our given function. Then we would have to derive what the T(n) is by running the function itself.
A way to do this could be to define a list of N's and double the value of N every time and compare the runtime for each set of N's.

Use stopwatch.java to measure your functions

Cost Frequency models will be important in this class for classifying runtime information. Another way to do it is to block your code into pieces by the frequency that they are used and figure out the runtime for each part and add it up.

Cost Frequency:

```
    1. Cost of the operation (time)
    2. Frequency (How often does it happen)
```

`summation of Cost(q)*Frequency(q)` q is the block of code that you are analyzing.

Lets build a Cost Frequency model for an example!

```
boolean isValid(String str){
    if(str.length < 2 || str.length % 2 != 0){ // this first block is only
executed once
        return false; // this is executed once
    }
    int mid = str.length/2; // this is executed once
    for(int 1 = 0; i < mid; i++){ // the init is executed once but the rest is
executed mutliple times
        if(str.charAt(i)!= 'a'|| str.CharAt(mid+i)!= 'b'){
            return false;
        }
    }
    return true;
}
```

We are splitting the code by the frequency, so the first section is stuff that is only happening once and the other block is the stuff that will happen multiple times. We will refer to the stuff that runs only once as A so A will only be executed once. We will refer to the stuff that runs multiple times as B and the frequency of how many times B will run is dependent on the length of the string. B executes n/2 times and A executes 1 time. Now we can build our cost frequency model. `T(n) = t0 + (t1*N/2)` is the runtime function for this example.

```
last_coin_standing(x,y)
Input: x number of quarters
             y position to eliminate

Output: last remaining quarters

quarters --> new array[x] // happens one time
for i <-- 1 to x do : // happens one time
    quarters[i] <-- i // happens N number of times
length <-- x // happens one time
pos <-- 0 // happens one time
while length > i do: // excutues n-1 times
    to_remove <-- pos + y - 1 // executes n-1 times
    to_remove <-- to_remove % length // executes n-1 times
copy_pos <-- 0 // executes n-1 times
new_array <-- new array[length - 1] // executes n-1 times
for i <-- 1 to length do: // occurs D number of times
    if j - 1 != to_remove then: // happens E number of times
        new_array[copy_pos] <-- quarters[j - 1] // happens E number of times
        copy_pos <-- copy_pos+1 // occurs D number of times
    quarters <-- new_array
    length <-- length-1
    pos <-- to_remove % length
return quarters[0] // happens one time
```

A: 1 (t0)
B: N (t1)
C: N−1 (t2)
D: (N(N + 1)/2) − 1 (t3)
E: (N(N − 1)/2) (t4)

`T(N) = t0 -t2 - t3 + N(t1 + t2 + (t3/2) - (t4/2)) + N^2((t4/2) + (t3/2))`
Our tilde approximation is: `~T(N) = N^2((t4 + t3)/2)`
We say some function f(n) has some tilde ~g(n) if `lim as N approaches infinity of f(n)/g(n) = 1`

9/20/16

Problems with HW 2:
1. If you mess with the signature of the equals you will not override the equals method

Polymorphism is the idea that you can overload methods.
beware of the `Instanceof` method. It can violate symmetry.

Lets do a frequency cost analysis of Homework #2:

```
Input: Keys - array of keys
Output: New awway with dupes removed

temp_array <-- new array[length]// only done once
for i <-- 1 to length do:
    temp_array[i] <-- key[i-1];// happens N number of times
sort(temp_array);// only done once BUT the java doc says that the worst case is
O(N*log(N))
pos <-- 1// only done once
for i <-- pos to length do:
    if key[i] != key[i-1] do:// happens N number of times - 1
        temp[pos] <-- key[i]// input dependent
        pos <-- pos +   1// input dependent
    a <-- new array[pos]// only done once
    for i <-- 1 to pos do:// only happens once
        a[i] <-- temp[i]
```

A: 1
B: N
C: N−1
D: X
E: Y
F: N*log(N) *Constant time*
T(N) = t0 + t1N + t2(N−1) + t3X + t4Y is the time function for this specific algorithm. By looking at this we can say that this is O(N).

Proof of induction

Time to learn proof of induction! O_O
1. The first step to this process is to establish a base case.
2. create a inductive hypothesis
a. prove it for k instances
b. k --> k + 1

Example with fibonacci numbers

```
F0 = 1, F1 = 1, F2 = 2, F3 = 3, F4 = 5 . . . . Fi = Fi-1 + Fi-2
i >= 1
Claim: For all i >= 1 Fib number Fi < (5/3)^i
Base case:
    i = 1
    F1 = (5/3)^1

    1 < (5/3) (This is true)
    F2 = 2 < (5/3)^2
    F2 = 2 < (25/9) (This is true)

Inductive Hypothesis:
    Assume Fi < (5/3)^i is true for all i <= k

We need to show that:
    F(k+1) < (5/3)^(k+1)

F(k+1) = F(k) + F(k-1)

Induction Step:
    F(k+1) < (5/3)^k + (5/3)^(k-1)
    F(k+1) < (3/5)(5/3)^k+1 + (9/25)*(5/3)^(k+1)
    F(k+1) < (3/5 + 9/25)(5/3)^(k+1)
    F(k+1) < (24/25)(5/3)^(k+1)
    F(k+1) < (5/3)^(k+1) --> we can do this because (24/25) is almost 1 and has
a small change to the main value
```

This is an example of a problem you might do in a generic math class. Below is an example of a problem we will see in this class

```
If N >= 1, then the sum from i=1 to N, i^2 = (N(N+1)(2N+1))/6
base case:
    let N = 1
    (1)^2 = 1 = ((1)((1)+1)(2(1)+1))/6 = 6/6 = 1

Assume the claim is true for 1 <= N <= k

Our induction hypothesis is:
    sum of i=0 to N+1 of i^2 = the sum from i=1 to N of i^2 + (N+1)^2
```

Proof of contradiction

```
1. Assume the opposite is true
2. Show some known property is false under this assumption
```

```
Claim: There are an infinite number of prime numbers
We assume that this claim is false E Pk where Pk is the largest prime
```

9/26/16

Dynamic Connectivity

Problem Specification

1. Input is a sequence of integer pairs where the integer represents some object (p, q)
2. We interpret a pair (p, q) as meaning p is connected to q
3. Connected?

- Reflective: (p, q)
- Symmetric: (p, q) (q, p)
- Transitive: (p, q) (q, r) (p, r)

4. Develop an algo that will filters pairs in the same equivalence class

**Example: **

On notepad

Union Find
It will be an abstract data type:

1. Initialization
2. Add connections between objects
3. Identify components/connections between objects (what equivalence class am I?)
4. Given x equivalence class, are you a member? (p, q belongs to X?)
5. Size

```
public class Abstract UnionFind
UnionFind (int N) //number of distinct object, size of our set

void union(int p, int q) //create a connection between p and q

int find(int p) //identify the component, how is it connected. which
component/class

boolean connected(int p, int q) // returns true if they are connected

int size() // number of distinct components, in our example 2 (2 diff distinct
connection going out)
```

Quick Find: Make find as quick as possible at any expense

- User an array to maintain the connection information
- id [p] == id [q] if p, q belongs to X

```
find (p)
Input: p integer representing object p
Output: integer component number of p

QuickFind:
  return id_array[p]; //constant time
```

Union has to traverse the array and transfer any id that is in same component as p

```
union (p, q, s) //performs linear time
Input: integer p and q representing objects p and q, s is the size
Ouput/Effect: p and q are connected (id_array[p] == id_array[q])

p_id <- find(p)
q_id <- find(q)

if p_id == q_id then:
  return s

N <- length of id_array
for i <- 0 to N -1 :
  if (id_array[i] == p_id) do:
    id_array[i] <- q_id

s <- s -1
return s
```

Example

starting array
0 1 2 3 4 5 6 7 8 9

---

input(4, 3)
0 1 2 3 3 5 6 7 8 9

---

input(3, 8)
0 1 2 8 8 5 6 7 8 9

---

input(6, 5)
0 1 2 8 8 5 5 7 8 9

For QuickFind we have N+3 or 2N + 1 time penalty for union --> linear in both cases

QuickUnion

```
find (p)
Input: p integer representing object p
Output: integer component number of p

while(p != id_array[p]) do:
  p <- id_array[p]

return p
```

9/28/16

- ID array of all unique objects
- Quick find
- Union operation is the drawback that limited us from calling this linear time

- Lets make union fast at the expense of the find.
  Below is an implementation of find

```
find(p, id)

input: object #
output: root/component # for object P (int)

while id[p] != p do:
    p <-- id[p]
return p
```

Below is an example of the Union method

```
Union(p,q,id)
Input: p/q: objects p and q to connect (integer)
Output: // updates linkage structure

P_root <-- find(p)
q_root <-- find(q)
if P_root = q_root then:
    return
id[p_root] <-- q_root
count <-- count-1
```

Time Analysis

Find: 1 + depth of tree (think of it as a binary tree)
Union: The absolute best case is O(5)

Algorithms
Most of the algorithms that we are going to be looking at this semester will be compare based algorithms. In order to produce a sorted result we will implement a comparison operation and act upon that.
Sorting algorithms are based on the idea of a key, java uses the comparable interface on the key to put everything in an order by an ordering rule.

```
Comparable<T>
int compareTo(T t);
```

We use compareTo to compare our data types based on their natural order

1. 0 = v.compareTo(w) when v == w
   -1 = v.compareTo(w) when v < w
   1 = v.compareTo(w) when v > w

2. v, w must be compatible with each other. You will get an exception if they are incomparable or either value is null.

3. Must be reflexive, transitive and symmetric

```
Public class Date implements comparable<Date>
    private int month;
    private int day;
    private int year;

    Public Date(m,d,y){
        month = m;
        day = d;
        year = y;
    }

int compareTo(Date obj){
    if(this.year > object.year){
        return 1;
    }
    if(this.year < object.year){
        return -1;
    }
    if(this.day > object.day){
        return 1;
    }
    if(this.day < object.day){
        return -1;
    }
    if(this.month > object.month){
        return 1;
    }
    if(this.month < object.month){
        return -1;
    }
    return 0;
}
```

```
Public class student implements comparable<student>{
    student (string fn, string ln);

    int compareTo(Student S){
        int val = this.ln.compareTo(s.ln);
        return val;
    }
}
```

10/3/16

`Enum Rank {two, Three .....}`

Knuth's algorithm randomizes elements in an array.

midterm on October 19th:
- 1/4 true or false
- 1/3 will me mechanics regarding algorithms (ie how does it work)

– remainder will be similar to the creative thinking problems on the homework.

Key – something that is comparable, could be irrelevant
Stability of an algorithm – A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted. Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. And some sorting algorithms are not, like Heap Sort, Quick Sort, etc.

Sorting algorithms

Selection Sort
The basic idea is that you iterate through the unsorted array and swapping the current position with the smallest integer emelment
Below is the pseudo-code for swap

Swap

```
Swap(a, i, j)
    input:
    a: array of items
    i, j --> positions to swap
    output: array with i, j swapped
    tmp <-- a[i]
    a[i] <-- a[j]
    a[j] <-- temp
```

Our boundary for Swap is O(N) for worst and best case
Below is the pseudo code for selection sort.

```
SelectionSort(a, n){
    a: array of items to sort
    n: length of array
    Output: natural order of items (keys)

    for i <-- 0 to N-1 do: //sorted partition of the array
        min <-- i
        for j <-- i+1 to N-1 do: // unsorted partition of the array
            if a[j] < a[min] then:
                min <-- j
            swap(a, i, min)
}
```

Best case and worst case for selection sort is O(N^2)

Insertion sort

If you have a stream of data coming in you can sort it with Insertion sort

```
InsertionSort(a, n)
    Input:
    a - array of items
    n - number of items

    Output: Sorted array

    for i <-- 0 to N-1 do:
        j <-- i
        while j > 0 && a[j] < a[j-1] do:
            Swap(a, j, j-1)
            j <-- j-1
```

Best case this is linear O(N) since if it is already sorted the while loop with finish in N time. Worst case this is O(N^2)

10/5/16

Suppose we want to improve upon Insertion sort where we want the increment to go back h amount of steps rather than one at a time:
By this we get,

Shell sort (h-sorting)

We introduce the notion of a gap sequence with shell sort
h = {4,2,1}
for the example above, it will h-sort by 4 then 2 then 1.
shell sort has a tilde time of, ~O(n*log(n))
CS people like it because it offers a lot of research material on how to optimize which gap sequencing scheme to pick.
You can almost get this to compete with quick sort with some of the published optimizations. The worst case also gets improved with these optimizations by reducing the exponent from a whole number such as 2 to 1.5.

```
0       1       2       3       4       5       6
T       I       N       Y       C       A       T   (h=4)
C       I       N       Y       T       A       T   (h=4)
C       A       N       Y       T       I       T   (h=4)
C       A       N       Y       T       I       T   (h=2)
C       A       N       I       T       Y       T (h=2)
A       C       N       I       T       Y       T   (h=1)
A       C       N       I       T       Y       T   (h=1)
A       C       I       N       T       Y       T   (h=1)
A       C       I       N       T       T       Y   (h=1)
```

There is a tradeoff with algorithms of memory and time. In order to improve in order of time we might have to take a hit in space. All the sorts we have gone over so far can all be done in the space that it is defined in. There is no need to duplicate or extend the arrays. Merge sort and quick sort are examples of sorting algorithms that require either recursion which takes up memory, or creating a copy of the array which takes up extra space.

If you have an int[] of size N you will lose 4N bytes of memory space because the size of all primitives in Java is 4 bytes

```
1. Harmonic Sum = 1 + 1/2 + 1/3 + 1/4 + 1/5 + ....... + 1/N ~O(log(n))
```

2. Triangular Sum = 1 + 2 + 3 + ... + N ~O((N^2)/2)
3. Geometric Sum = 1 + 2 + 4 + 8 + ... + N = ~O(2N-1)
4. N choose K = ~O((N^K)/K!)
   5.Exponential approximation = (1-1/x)^x = ~O(1/e)

Merge Operation and Merge Sort

Merge: takes 2 ordered arrays --> combines into 1 ordered array

```
Input = a = [2, 4, 6, 8]
                b = [1, 3, 5, 7]
Output = c = [1, 2, 3, 4, 5, 6, 7, 8]
```

Pseudo code of merge sort

```
abstract_Merge(a[], low, mid, high)
    input: array of keys, with 2 ordered partitions defined by low, mid, high
                 low: beginning of first partition
                 mid: end of first partition
                 high: end of the second partition

    i <-- low
    j <-- mid+1
    for k <-- low to high do:
        aux[k] <-- a[k]
    for k <-- low to high do:
        if i > mid then
            a[k] <-- aux[j]
            j <-- j+1
        else if i < mid then
            a[k] <-- aux[i]
            i <-- i+1
        else if aux[j] < aux[i] then
            a[k] <-- aux[j]
            j <-- j+1
        else do:
            a[k] <-- aux[i]
            i <-- i+1
```

Example of Merge sort:

```
a[] = [a,d,f,g,w,a,b,e,f,m]
merge(a,0,4,9)
        K       I       J       a,d,f,g,w | ,a,b,e,f,m
        0       1       5       a,d,f,g,w | ,a,b,e,f,m
        1       1       6       a,a,f,g,w | ,a,b,e,f,m
        2       1       7       a,a,b,g,w | ,a,b,e,f,m
        3       2       7       a,a,b,d,w | ,a,b,e,f,m
        4       2       8       a,a,b,d,e | ,a,b,e,f,m
```

10/10/16

```
Merge (a, lo, mid, hi)
    Input: a: array of items s.t. the partitions defined by lo, mid and hi are
sorted
        Lo : start of first partition
        Mid : end of 1st partition
        Hi: end of 2nd partition

    Output: Partitions combined into one sorted partition

    I <- lo
    J <- mid + 1
    Aux <- new_array
    For k <- lo to hi do:
        Aux [k] <- a[k]

    For k <- lo to hi do:
        If i > mid then:
            A[k] <- aux[j]
            J <- j+1
        Else if j > hi then:
            A[k] <- aux [i]
            I < i+1
        Else if aux[j] < aux[i] then:
            A[k] < aux [j]
            J <- j+1
        Else do:
            A[k] <- aux[i]
            I <- i+1
```

Divide and conquer idea
First loop creates working copy
The merge operation requires additional memory
Because the merge sort is based on a merge operation it's not an in place sort

Top Down MergeSort
Sort (a, lo, hi)
Input: a – an array to sort
Lo – 1st element
Hi – last element

This has a time classification of Nlog(N) which is no better than any of the other methods we have gone over so far.

10/13/16

Chapters 2.1, 2.2, 2.3 to study for the midterm

Nlog(N) Barrier

Claim: No compare based sorting algorithm can guarantee to sort N items with fewer than Nlog(N) compares

Proof:

```
Assume all keys are distinct
Define the notion of a compare tree
The Compare Tree is a binary tree with some additional rules:
    - Each leaf node represents a completed sort outcome for your array
    - Each internal node represents a compare operation on the pair (i,j)
compare (a[i],a[j])
    - Left subtrees represent --> sequence of compares where a[i] was less than
a[j]
    - Right subtrees represent --> sequence of compares where a[i] was greater
than a[j]
    - From all of these assumptions we can say that each path from the root to
a leaf node represents the sequence of compares to produce the outcome at the
leaf node
    - we can see from the compare tree that there are N! ways of arranging
these keys, this also means we have N! sort operation outcomes
    - we can see from the compare tree that the best case is the leaf on the
far left of the tree and the worst case is the leaf on the far Right
    - The height of the tree is the worst case of the sort
    - if we have a binary tree of height h we know that it has no more that
2<sup>h</sup> nodes
    - Every compare tree has some a height h and some number of leaf nodes that
are between N! and 2<sup>h</sup>

N! <= 2<sup>h</sup> and N! < l <= 2<sup>h</sup>

log(!N) <= log(2<sup>h</sup>)

log(N!) <= h

log(1)+.....log(N)

~ Nlog(N) <= h

~ Nlog(N) < l <= h

we have shown for this general case that we cannot do better than Nlog(N)
```

A priori key knowledge

Shannon Entropy: N keys with k distinct values
for each i from 1 to K
F_i : frequency of key K_i
P_i : frequency of that key divided by the number of keys which is the probability of key Ki
H = Shannon entropy which is the negative sum of the keys from i = 1 to K P_i * log(P_i)
In the worst case when these are distinct this will collapse into an Nlog(N) time magnitude
In the best case when these are the same this will be of order 1 time magnitude

Quick sort

lets build an algorithm that can take advantage of the non uniqueness of keys
Quick Sort uses a partitioning system to sort the elements

Partition
A partition re-arranges an array to enforce the following rules
1. One value is a[j] is in its final place
2. No entry from a[min] up to a[j-1] is greater than the value at a[j]
3. No entry from a[j+1] to a[max] is less than the value at a[j]
4. we call that value that is in its final place the pivot

```
A = {T I N Y C A T}
assume pivot is A[0]
where the resulting array is {C I N T T A Y}

- this is not a partitioning function because the value C is not at its final
spot
```

```
A = {T I N Y C A T}
assume pivot is A[0]
where the resulting array is
{C I N A T T Y}

- this is a partition because the T is in its final spot and everything after
the T is greater than it and everything before is is less than T
```

```
A = {T I N Y C A T}
assume pivot is A[3]
where the resulting array is
{T I N T C A Y}

- This partition is correct even though the pivot is in the far right spot.
```

Pseudo Code for Quick Sort

```
quicksort(A, lo, hi)
    input:  A - array
                    lo - starting position
                    hi - end position

    if lo < hi then
      p := partition(A, lo, hi)
      quicksort(A, lo, p – 1)
      quicksort(A, p + 1, hi)

partition(A, lo, hi)
  pivot := A[hi]
  i := lo        // place for swapping
  for j := lo to hi – 1 do
      if A[j] ≤ pivot then
          swap A[i] with A[j]
          i := i + 1
  swap A[i] with A[hi]
  return i
```

10/17/16

Continuing on Quick Sort.

Below is an example of a method that returns a partition point for quick sort.

```java
private static int partition(Comparable[] a, int lo, int hi){
    int i = lo, j = hi+1;
    Comparable v = a[lo];
    while (true){
        while(less(a[++i], v)) if(i == hi) break;
        while(less(v, a[--j])) if(j == lo) break;
        if(i >= j) break;
        exch(a, i, j);
    }
    // this final exchange moves our pivot into place
    exch(a, lo, j);
    return j;
}
```

Steps for deciding where to make the partition of the array

```
1. let C<sub>N</sub> be the average # of compares (Nlog(N)) needed to sort N
distinct keys
2. C<sub>0</sub> = C<sub>1</sub> = 0
3. N + 1 = partition
4. 1/N * (The sum from n=0 to N-1 of C<sub>n</sub>)
5. C<sub>N</sub> = N + 1 + 2/N * (The sum from n=0 to N-1 of C<sub>n</sub>)
```

For the test go over generic Java concepts

1.
– Generics
– Type erasure
– Various common interfaces:
– Comparable
– Iterator
– Iterable
– mutable vs. immutable types
– Basic data types:
– bags (unordered collection)
– queues (Gives you the ability to enqueue and dequeue FIFO)
– stacks (Push Pop operations, LIFO)
– Key data structures/the abstract data types
– static vs. non static access
2.
– Analysis:
– big O order of growth functions
– Definition of what the Big O function is
– Can you do a cost frequency Analysis
– Build algorithm X from pseudocode that performs in O(something)
– True or false questions based on their respective big O notations
3.
– Sorting:
– working with modifying, comparing and analyzing sorting algorithms
– be familiar with:
– insertion
– selection
– merge
– quick
– h sort (shell sort)

Reading Notes

Chapter 2.1

Our primary concern when it comes to algorithms, is to rearrange arrays of items where each item contains a key. The idea behind rearranging the keys is to do to is a manner so that the keys are naturally ordered.

We put our sort code in a sort() method within a single class along with private helper functions `less()` and exch(). `less()` checks whether a number is less than the one it is being compared to and exch() will swap two values in an array.

The comparable interface makes implementing `less()` very straightforward

Certification

Does sort implementation always pit the array in order, no matter what the initial order? As a conservative practice, we include the statement `assert isSorted(a);` in our test client to certify that array entries are in order after the sort. It is reasonable to include this statement in every sort implementation, even though we normally test our code and develop mathematical arguments that our algorithms are correct.

Extra Memory The amount of extra memory used by a sorting algorithm is often as important a factor as running time. The sorting algorithms divide into two basic types: those that sort in place and use no extra memory except perhaps for a small function-call stack to a constant number of instance variables, an those that need enough extra memory to hold another copy of the array to be sorted.

Types of Data

Our sort code is effective for ant item type that implements the `Comparable` interface. Adhering to Java's convention in this way is convenient because many of the types of data that you might want to sort implement `Comparable`. For example, Java's numeric wrapper types such as Integer and Double implement Comparable, as do String and even more complex variable types such as FILE and URL. Listed below is an example of the `compareTo` method.

```
public int compareTo(Date that){
    if(this.year > that.year) return +1;
    if(this.year < that.year) return -1;
    if(this.month > that.month) return +1;
    if(this.month < that.month) return -1;
    if(this.day > that.day) return +1;
    if(this.day < that.day) return -1;
}
```

Java's convention is that the call `v.compareTo(w)` returns a integer that is negative, zero, or positive when v < w, v = w, or v > w, respectively. By convention, code `v.compareTo(w)` throws an exception if v and w are incompatible types or either is null. Furthermore `compareTo()` must implement a total order. it must be
1. Reflexive (for all v, v = v)
2. Antisymmetric (for all v and w, if v < w then w > v and if v=w then w=v)
3. Transitive (for all v, w, and x, if v <= w and w <= x then v <= x)
These rules are intuitive and standard in mathematics.

Selection Sort

one of the simplest sorting algorithms is as follows, find the smallest element in the array and exchange it for the first value in the array. Then find the next smallest element and exchange it for the second value in the array. Continue this until the entire array is sorted. This method of sorting is called selection sort and is one of the most trivial sorting algorithms out there. Below is an example of selection sort

```
public class Selection{
    public static void sort(Comparable[] a){ // sort a into increasing order
        int N = a.length;
        for(int i = 0; i < N; i++){ // Exchange a[i] with smallest entry in
a[i+1...N).
            int min = i; // index of a minimal entry
            for(int j = i+1; j < N; j++){
                if(Less(a[j], a[min])) min = j;
            exch(a, i, min);
            }
        }
    }
}
```

The work of moving the items around falls outside the inner loop: each exchange puts an item into its final position so the number of exchanges in N. Thus, the running time is dominated by the number of compares.

Proposition A
Selection sort uses $N^2/2$ compares and N exchanges to sort an array of length N.

You can prove this with by examining the trace table and noticing that the table in N by N in which unshaded letters correspond to compares.

i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X

In summary selection sort is a simple sorting method that is easy to understand and to implement and is characterized by the following two signature properties.

Running time is insensitive to input
The process of finding the smallest item on one pass through the array does not give much information about where the smallest item might be on the next pass .This property can be disadvantageous in some situations. For example, the person using the sort client might be surprised to realize that it takes about as long to run selection sort for an array that is already in sorted order or for an array with all keys equal to as it does for a randomly ordered array.

Data movement is minimal

Each of the N exchanges changes the value of two array entries, so selection sort uses N exchanges. the number of exchanges is a linear function of the array size. None of the other sorting algorithms that we consider have this property. (most involve linearithmic or quadratic growth)

Insertion Sort

The algorithm that people often use to sort bridge hands is to consider the cards one at a time, inserting each into its proper place among those already considered. In a computer implementation we need to make space to insert the current item into the vacated position. This method is called insertion sort. Code for this can be seen below.

```
public class Insertion{
   public static void sort(comparable[] a){
       int N = a.length;
       for (int i = 0; i < N; i++){
           for(int j = i; j > 0 && less(a[j], a[j-1]); j--){
               exch(a, j, j-1);
           }
       }
   }
}
```

For each i from 1 to N−1, exchange a[i] with the entries that are larger in a[0] through a[i−1]. As the index i travels from left to right, the entries to its left are in sorted order in the array, so the array is fully sorted when i reaches the end right.

Unlike selection sort, the running tie of insertion sort depends on the initial order of the items in the input. For example, if the array is large and its entries are already in order (or nearly in order), then insertion sort is much, much faster that if entries are randomly ordered or in reverse order.

Proposition B
Insertion sort uses ~$N^2/4$ compares and ~$N^2/4$ exchanges to sort a randomly ordered array of length N with distinct keys, on the average. The worst case is ~$N^2/2$ compares and ~$N^2/2$ exchanges and the best case is N−1 compares and 0 exchanges.

i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

Proof B:
the number of compares and exchanges is easy to visualize in the N by N diagram that we use to illustrate the sort. We count entries below the diagonal -- all of them, in the worst case, and none of them, in the best case. For randomly ordered arrays, we expect each item to go about halfway back on the average, so we count one-half entries below the diagonal. The number of compares is

the number of exchanges plus an additional term equal to N minus the number of times the item is inserted is the smallest so far. In the worst case (array is reverse order), this term is negligible in relation to the total; in the best case it is equal to N-1.

Insertion sort works we;; for certain types of nonrandom arrays that often arise in practice, even if they are huge. For example, if you already have a sorted array that you implement insertion sort on, the sort will immediately determine that each element is in its correct place and the total running time is linear. The same is true for an array who's keys are all equal.

Shell Sort

To exhibit the value of knowing properties of elementary sorts, we next consider a fast algorithm based on insertion sort. Insertion sort is slow for large unordered arrays because the only exchange it does involves adjacent entries, so items can move through the array only one place at a time. For example if the time with the smallest key happens to be at the end of the array N-1 exchanges are needed to get that one item where it belongs, shellsort is a simple extension of insertion sort that gains speed by allowing exchanges of array entries that are far apart, to produce partially sorted arrays that can be efficiently sorted, eventually by insertion sort.

The idea is to rearrange the array to give it the property that taking every Hth entry yields a sorted subsequence. Such an array is said to be h-sorted. Put another way, an h-sorted array is h independent sorted subsequences, interleaved together. By h-sorting for some large value of h, we can move items in the array long distances and thus make is easier to h-sort for smaller values of h. Using such a procedure for any sequence of values of h that end in 1 will produce a sorted array: that is shellsort. An example of shellsort can be found below.

```
public class Shell{
    public static void sort(Comparable[] a){
        int N = a.length;
        int h = 1;
        while(h < N/3) h = 3*h + 1;
        while(h >= 1){
            for(int i = h; i < N; h++){
                for(int j = i; j >= h && less(a[j], a[j-h]); j -= h){
                    exch(a, j, j-h);
                }
            }
            h = h/3;
        }
    }
}
```

The general convention for using shell sort is using an h sequence of decreasing values $1/2*(3^k-1)$, starting at the smallest increment greater than or equal to N/3 and decreasing to 1. we refer to such a sequence as an increment sequence. The example above computes its increment sequence; another alternative is to store an increment sequence in the array.

One way to implement shell sort would be, for each h, to use insertion sort independently on each of the h subsequences. Because the subsequences are independent, we can use an even simpler approach: when h-sorting the array, we insert each item among the previous items in its h-subsequence by exchanging it with those that have larger keys (moving them each one position to the right of the sequence). We accomplish this by implementing insertion– sort code, but modified to decrement by h instead of 1 when moving through the array. This observation reduces the shell sort implementation to an insertion sort like pass through the array for each increment.

Shell sort gains efficiency by making a tradeoff between size and partial order in the subsequences. At the beginning, the subsequences are short; later in the sort the subsequences are partially sorted. In both cases, insertion sort is the method of choice. The extend to which the subsequences are partially sorted is a variable factor that depends strongly on the increment sequence.

As you can learn from comparing insertion sort and selection sort, shell sort is muc faster that insertion sort and selection sort, and its speed advantages increases with the array size. Shell sort makes it possible to address sorting problems that could not be addressed with the more elementary algorithms.

Chapter 2.2 Merge Sort

The algorithms described in this chapter are based on combining two ordered arrays to make one larger ordered array. This operation immediately leads to a simple recursive sort method known as **merge sort**; to sort an array, divide it into two halves, sort the two halves (recursively) and then merge the results. One of the major benefits of merge sort is that it guarantees to sort any array of N elements in time proportional to Nlog(N).

Abstract in place merge.

The straightforward approach to implementing merging is to design a method that merges two disjoint ordered arrays of comparable objects into a third array. This strategy is easy to implement: create an output array of the requisite size and then choose successively the smallest remaining item from the two input arrays to e the next item added to the output array.

However when we merge sort a large array, we are doing a huge number of merges, so the cost of creating a new array to gold the output every time that we do a merge is problematic. This is where in place merge comes in to play. The idea behind this is that we sort the first half then sort the next half and merge the two halves. Solutions to this tend to be quite complex especially by comparison to alternatives that use extra space. Below is source code for an implementation of this.

```
private static void merge(Comparable[] a, int lo, int mid, int hi){
    int i = lo, j = mid+1;

    for(int k = lo; k<= hi; k++){
        aux[k] = a[k];
    }
    for (int k = 0; k <= hi; k++){
        if (i > mid)                                    a[k] = aux[j++];
        else if (j > hi)                            a[k] = aux[i++];
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
        else                                                a[k] = aux[i++];
    }
}
```

Top Down MergeSort

This algorithm of Top Down merge sort is one of the best examples of a divide and conquer paradigm for efficient algorithm design.

To understand merge sort, it is worthwhile to consider carefully the dynamics of the method calls. The idea behind this is recursively call the sort method to sort smaller and smaller subsets of the array then once the two are successfully sorted for their respective halves, there is one major sort that will sort the two halves together. The benefit of this is that when you call merge the more computationally difficult two halves, then those halves are already sorted so the process becomes a lot easier for a computer to handle. Top-down merge sort uses between ~1/2Nlog(N) and Nlog(N) compares to sort any array of length N.

We can improve most recursive algorithms by handling small cases differently, because the recursion guarantees that the method will be used often for small cases, so improvements in handling them lead to improvements in the whole algorithm. In the case of sorting, we know that insertion sort or selection sort is simple and therefore likely to be faster than merge sort for tiny subarrays. Another improvemnet we coudl make is to add a test to call the skip to `merge()` if a[mid] is less than or equal to a[mid+1. With this change we still do the recursive calls but the running time for sorted arrays in linear.

Chapter 2.1 Quick sort

One of the must popular sorting algorithms out there is known as quick sort. Quick sort is popular among the community because it is not difficult to implement, works well for a variety of different kinds of input data, and is substantially faster than any other sorting method in typical applications. The quick sort algorithm's desirable features are that it is in-place and that it requires time proportional to N*log(N) on the average to sort an array of length N. None of those algorithms that we have covered so far share these two properties. Also quick sort has a shorter inner loop than most other sorting algorithms, which means that it is faster in practice and theory.

One of the few downsides of quick sort is that it is fragile in the sense that some care is involved in the implementation to be sure to avoid bad performance. Numerous mistakes in taking care of how to handle quick sort can quickly lead to it being quadratic.

Quick sort is a divide and conquer method for sorting. It works by partitioning an array into sub-arrays, then sorting the subarrays independently. Quick sort is complementary to merge sort, expect for quick sort we arrange it so that when we partition the two halves and put them back together the entire array is sorted. Below is example code for quick sort.

```
public class Quick{
    public static void sort(comparable[] a){
        StdRandom.shuffle(a);
        sort(a, 0, a.length -1);
    }

    private static void sort(Comparable[] a, int lo, int hi){
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
    }
}
```

In the first instance we do the two recursive calls before working on the whole array; in the second instance, we do the two recursive calls after working on the whole array. For quicksort the position of the partition depends on the contents of the array.

The crux of the method is the partitioning process, which rearranges the array to make the following three conditions hold:
1. The entry a[j] is in its final place in the array, for some j
2. No entry in a[lo] through a[j-1] is greater than a[j]
3. no entry in a[j+1] through a[hi] is less than a[j]
We achieve a complete sort by partitioning, the recursively applying this method. Because the partitioning process always fixes one item into its position, it is relatively easy to produce a proof by induction.

Partitioning in place

If we use an extra array, partitioning is easy to implement, but not so much easier that it is worth the extra cost of copying the partitioned version back into the original.

Staying in bounds

If the smallest item or the largest item in the array is the partitioning item, we have to take care that the pointers do not run off the left or right ends of the array. Our Partition() implementation has explicit tests to guard against this circumstance. the test (j == lo) is redundant, since the partitioning item is at a[lo] and not less that itself.

Preserving randomness

The random shuffle puts the array in a random order. Since it treats all items i the subarrays uniformly. This fact is crucial to the predictability of the algorithm's running time. An alternate way to preserve randomness is to choose a random item for partitioning within partition.

Helpful hints for the midterm

Well, in Java an int is a primitive while an Integer is an Object. Meaning, if you made a new Integer:

```
Integer i = new Integer(6);
```
You could call some method on i:

```
String s = i.toString();//sets s the string representation of i
```
Whereas with an int:

```
int i = 6;
```
You cannot call any methods on it, because it is simply a primitive. So:

```
String s = i.toString();//will not work!!!
```

---

markdown-pdf AlgorithmNotes.md

Various Big-O Notations

Algorithm	**Time Complexity**			**Space Complexity**
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Static Method

Static methods are called `functions` in many programming languages. The modifier static distinguishes these methods from instance methods which must be called on a object so a.sort();

Abstract Data Types

To specify the behavior of an abstract data type, we use an API, which is a list of constructors and instance methods, with an informal description of the effect of each, as in the API for counter

```
public class counter


                Counter(String id) // create a counter named id
void        increment()              // increment the counter by one
int         tally()                      // number of increments since creation
String  toString()               // string representation
```

Inhereted methods

All objects in java must inherit a toString() method. If we use the general toString that it inherits we would get a string with the memory representation so generally the user would override the method and implement their own. Other examples of methods that are generally overided are equals(), compareTo(), hashCode().

Objects

Naturally, you can declare that a variables heads is to be associated with data of type Counter with the code `Counter heads;`. An object is an entitty that can take on a data-type value. Objects are characterized by three essential properties: state, identity, behavior. The state of an object is a value from a data type. The identity od an object distinguishes one object from another. It is useful to think of an objects identity as a place where it is stored in memory. The behavior of an object is the effect of data-type operations.

Creating Objects

Each data-type value is stored in an object. To create an individual object, we invoke a constructor by using the keyword new, followed by the class name, followed by () (or a list or args). A constructor has no return type because it always returns a reference to an object of its data type. Each time that a client uses new(), the system
1. Allocates memory space for the object
2. invokes the constructor to initialize its value
3. REturns a reference to the object

In client code we typically create objects in an initializing declaration that associates a variable with an object, as we often do with variables of primitive types. Unlike primitive types, variables are associated with references to objects, not the data-type values themselves.

Equality of Objects

IF we test objects with (a == b) where a and b are reference variables of the same type we are testing whether they have the same identity: whether the references are equal. An example of using the .equals instance method, if x and y are string values, then x.equals(y) is true if and only if x and y have the same length and are identical in each character position. Java's convention is that equals() must be an equivalence relation. It must be
1. Refelxive, x.equals(y) is true
2. Symmetric, x.equals(y) is true and y.equals(x) is true
3. Transitive, if x.equals(y) and y.equals(z) are true then x.equals(z) is true

```
public boolean equals(Object x){
    if (this == x) return true;
    if (x == null) return false;
    if (this.getClass() != this.getClass()) return false;
    Date that = (Date) x;
    if (this.day != that.day) return false;
    if (this.month != that.month) return false;
    if (this.year != this.year) return false;
    return true;
}
```

Immutability

An immutable data type such as date has a property that the value of an object never changes once constructed. By contrast a mutable data type such as a Counter or accumulator manipulates object values that are intended to change. Java supports this feature with the final modifier. When you declare a variable to be final, you are promising to assign it a value only once.