http://bigocheatsheet.com/

CS 314 Study Guide

- **Big O Cheat Sheet:** here
- Sample Questions Quizlet

Big O - An upper bound of how quickly a program will execute..

$1 < \log(n) < \log^2(n) < n < n * \log(n) < n^2 < n^k < 2^n < 3^n < k^n < n! < n^n$

Ratio	Power of 2	Big O
2	2^1	O(n)
2+	2^1+	O(n(log(n)))
4	2^2	$O(n^2)$
8	2^3	$O(n^3)$

*Note about ratio: Ratios = (runtime of 2n) / (runtime of n)

- Best sorting algorithms = O(n*log n) (other than radix sort or bucket sort)
- Search of an unsorted list/array/etc= O(n)
- Elementary functions, array access are O(1) --> +,-,/,*
- A sequence of statements is the max Big O of the statements
- For an if statement it is the max of the test, then statement, and else statement
- Loop is the loop count times the Big O of the contents
- Binary Search O(logn) (discard one half of a recursive statement)
- Quicksort O(nlogn) or (n^2) (process both halves of a recursive statement)
- Tree search with discarding branches O(logn) (binary search tree, aka sorted)
- Tree search with using all limbs O(n logn) (unsorted)

  Search in sorted array O(logn)

Log-Log graph: y-axis and x-axis are logarithmic
if a graph is linear in log-log graph, it is **polynomial** time ($\mathbf{x^n}$ e.g. x^2)
if it curves upwards in log-log graph, it is exponential (2^n) and intractable

Semi log graph: y-axis is logarithmic, x-axis is linear
if a graph is linear in a semi-log graph, it is 2^n (exponential) and is **intractable**

1 byte: 8 bits   10 bits: 3 decimal
int: 32 bits (4 bytes) (9 decimal)
Integer: 128 bits (16 bytes)
double: 64 bits (8 bytes)
long: 64 bits (not used as much as double)   (19 decimal)
float: 32 bits (don't use!)

Big O of loops: multiply by n every loop
O(n):
```
int[] myarray = new int[1000];
for (int i = 0; i < 1000; i++)
       myarray[i] = 0;
```

O(n^2):
```
int[][] myarray = new int[1000][1000];
for (int i = 0; i < 1000; i++)
       for (int j =i; j < 1000; j++)
              myarray[i][j] = 0;
```

a “bermuda triangle”, which is (1/2) n^2 = O(n^2)
-
--

Pointer/reference: memory address of where the contents are stored
An Object in java contains a pointer
Integer i = 0; Integer j = i;   i++;
i++ creates a new integer in memory. Integers are IMMUTABLE. You can’t increment and decrement the actual values in the integer memory “box”. So i++ would basically create a new Integer value in another location while the pointer “J” is still pointing to the original integer memory “box”.
TL;DR: If you do i++, i = 1 and j = 0.
i++ ⇔ Integer i = new Integer(1); ⇔ Integer i = Integer.valueOf(1);
The original i’s memory location is no longer referred to by Integer i, however it is not garbage collected because it is still referenced by Integer j.

A pointer is often **8 bytes** (64-bit machines)

Boxed number: have same numeric accuracy as primitive type, but need more memory since it’s an **Object** (e.g. **Integer (16 bytes)** vs int (4 bytes)), often capitalized (Double vs double)
Integer (and other boxed numbers) is **immutable** e.g. Integer x = 5; x += 1; adding one to x creates a **new** Integer of 6 instead of “adding” one to the old Integer (unlike int) - the old Integer becomes **Garbage** (unpointed to and sometimes “garbage collected”)
note: Java has a cache of small Integers, so it’s more efficient to use *Integer.valueOf(x)* over *new Integer(x)*, if x is a small #
The range of the number is [-127,127].

Array of Objects only stores the pointers - the actual contents of the array are stored elsewhere e.g.
```
Integer[] myarray = new Integer[1000];
for (int i = 0; i < 1000; i++){
 myarray[i] = new Integer(i);
}
```

size of myarray: 8 bytes (myarray is only a name for the pointer -> 8 bytes)
size of the objects stored in array: 8000 bytes (myarray only stores pointers -> 8 * 1000 = 8000 bytes)
total size allocated for array: 24000 bytes (the actual data is stored elsewhere; 16 (size for Integer) * 1000 + 8000 from pointers in array = 24000 bytes)

However, array of primitive objects actually stores the whole primitive type inside (e.g. array of int[1000] is 4000 bytes)

== measures pointer equality, while .equals() measures equality of contents
e.g.
Cons a = list("x");
Cons b = list("x");
a == b False
a.equals(b) True

Use .equals() for most cases unless it's a primitive type (int, double), for pointers, or null

"Do not use == to compare values for any reference (Capital-letter) type in Java. Use .equals() ." - Novak

Cons

Cons is a singly-linked linked list- it consists of two fields: link/pointer to next field (or null), and contents

cons(a, b) links Object a to
lisp: (list 'a 'b 'c)

"cons"ing "a" to (b, c) (makes (a, b, c)):
java: cons("a", list("b", "c"))
lisp: (cons 'a '(b, c))

first: returns first object in cons
ex. (first '(a b c)) -> a
(first '((a b) c (d))) -> (a b)

first() returns an **Object** as declared type (compiler); however**,** the **runtime type** will be whatever the reference type is
if you call first(), sometimes you need to **cast** since Java is **stupid** and only reads the thing returned as an Object, not necessarily its actual type.*

> **That's true of pretty much all strongly typed languages and it's not actually that stupid - on the contrary, it makes the code safer to run. This "issue" can be rectified with the correct use of generics, but then you're limited to one return type.*

e.g. Integer x = (Integer) first(list); - the value returned by first(list) is an Object
first returns a Object (ex. it can only return boxed numbers like **Integer** and **Double** instead of primitive types like int and double)

rest: returns rest of the cons minus the first
ex. (rest '(a b c)) -> (b c)
(rest '((a b) c (d))) -> (c (d))
rest() returns a **Cons**

you can call rest multiple times; if rest is null, it returns null/nil
second: first(rest(a))
third: first(rest(rest(a)))   and so on…

Some Cons methods: **All of these methods are O(n)**
reverse(a) - reverses the order of list; constructive -O(n) // reverse(reverse(a)) - creates a copy of the list
append(a, b) - concatenates two lists to form single list -O(n) where n is length of **first** list a (reuses second list)
length(a) - returns length of list - O(n)
copy(a) - copies list (basically calls reverse(a) twice) - use in order to have a new copy instead of just a pointer -O(n)

nreverse(a,b) - reverses a list in place by turning the pointers around --- Big-O: O(n)

Iteration: walk along a list and do something

```
for (Cons lst = x; lst != null; lst = rest(lst)) {do something}
```

Recursion: calls itself as a subroutine

1. test for **base case** answer
2. Otherwise, **case**(s), and compute/return value directly (may include safe
    a. calls itself recursively to do **smaller** parts
    b. compute answer in terms of the smaller parts

Recursive calls always involve smaller arguments: **well-founded ordering** (ordering that can be guaranteed to terminate)
-otherwise it can go in infinite loop`

Tail recursive processing: calls another function that is the same thing as itself except it has extra parameters

```
public static int count(Cons a) {
       return countb(a, 0);
}
public static int countb(Cons a, int answerholder) {
       do the actual work in this method, use the extra parameter to store the answer  ("helper method")
}
```

If your compiler is smart, it'll recognize and compile tail-recursion as iterative to use only *o(1)* stack space instead of say, *o(n)* stack space (usually short)

** Note: You can do tail recursion without an auxiliary function.*

Constructive method: a method that does not modify the original object destructively (e.g. does not use any methods with "set" for Cons)

Destructive method:

- modifies original object destructively (e.g. uses a method with "set")
- often starts with a "d" or "N" like dmerj
- have to save value: e.g. mergedlist= dmerj(mergedlist); -
    - don't just call dmerj(mergedlist); (not "dropping the ball")
- it's better to use constructive than destructive methods- destructive methods can cause unintended consequences since other things may point to it; however, they may use less computer resources/memory/ be in-place

Merge (not Mergesort): walk down two **sorted** lists simultaneously and comparing the top values, and adding the values to a third list according to how the two top values compare as it walks down. Duplicates are retained. (**Both lists have to be sorted first!!!**)

Sets: member, assoc, intersection, some, every, subset, set-difference, union (look later down on review for definitions/examples)

Comparison in Java:
primitive types: <, >, == (e.g. int, double)
-cannot use .compareTo()

Objects (including Strings): use .compareTo()
if a.compareTo(b) == 0 then a.equals(b)
if a.compareTo(b) <0 then a is smaller than b
if a.compareTo(b) >0 then a is bigger than b
It pretty much doe s a-b and then give you a value

you can write a **Comparator** or "complex" Comparator to have a custom comparison method (e.g. for classes you write yourself

Divide and Conquer: solve large problem by breaking it down into two smaller problems, and solving those

-nlogn and logn time
-cut in half and process one half: **O(logn)** (like binary search)
-cut in half and process both halves: **O(nlogn)** (like sorting)

Dividing list/finding midpoint: **O(n)**
-keep two pointers, and move the second pointer two steps each time the first pointer moves 1 step

Mergesort:
```
if (list is length 0 or 1) {
      return the list
} else {
      break the list into two halves by using midpoint
      merge(mergesort(first half), mergesort(second half))
}
```

remember merge is not the same thing as mergesort!

For-each loop
```
      for (Object myValue : myCollection) {
            do something with myValue;
      }
```
it's really just shorthand for using the Iterator, so the big O is based off of that.

interface
Arrays:
Random access, access to any element is O(1)
Rigid, cannot be expanded (requires you to create a new larger array) -like adding to a brick wall
Storage may be wasted because an array is made larger than necessary to accommodate the data you need.
Sparse arrays - most of the elements are zero or missing. All real world graphs are sparse.

Trees
a tree is a type of graph composed of links and nodes
- one node, called root, has no incoming links
- each node has **exactly one** "parent"
- every node is reachable from root
- node with children: interior node; no children: leaf

First Child/Next Sibling Tree (for linked lists)
what are the advantages of first child/next sibling?
- Can support a large amount of children (isn't limited to 2 like, for example, in a binary tree)
- Requires less memory
- Remember Gordon Shaw Novak's saying: "same taste, less filling - beer"

Every "node" of a (binary) tree is composed of 3 parts: the parent, the left child, and the right child
parent (itself): first(x)   (op(x)) - is always an Object (never points to Cons)
left child: second(x)   (lhs(x)) -can be interior node or leaf
right child: third(x)   (rhs(x)) - can be interior node or leaf
Every time you traverse to another node, you check if it's a Cons (consp), and if it is, cast to a Cons (is another part of tree),otherwise it's a leaf and cast to that type stored

Binary Search Tree: A tree such that it is sorted so that every node to its left has descendents less than its contents, and every node to right has descendents greater than contents

- use for binary search if sorted

```
if (num == node's contents )
        return contents
 else if(op.compare(lhs) < 0)
        go left
else
        go right
```

Pre, in, post order:

Preorder processes the parent in front of the children. Used in directory navigation. <root><left><right>
Inorder processes the parent in between the children. Used to print infix expressions like the ones in Java, print sorted binary trees, flatten into ordered list by reverse inorder traversal. <left><root><right>
Postorder processes the parent after the children. Used in function eval, evaluating an expression tree, <left><right><root>
Example: 5 * x
  Preorder: (* 5 x)
  Inorder: (5 * x)
  Postorder: (5 x *)

Easy way to remember: use the root (if preorder - root is first; postorder - root is last; inorder - root is in the middle), and put in the left before the right child once you have this.

Sets:

Intersection - Elements that are in both sets.

- (1,2,4) ∩ (1,3,6,4) = (1,4)

Union - Elements that are in either set.

- (1,2,3) U (1,4,3) = (1,2,3,4) //No duplicates for union

Set difference - Elements that are in the first set, but not in the second set.

- (a b c) - (a c e) = (b) //Remember, just looking at the first
- (a c) - (a c b) = () //Empty set because all elements in first set exist in second

Member: member(Object item, Cons lst) → returns null if requested item isn't found

- returns a cons of the item + **all the items after it**
- (member 'dick '(tom dick harry)) = (dick harry) ← haha ← this ←

Assoc: assoc(Object key, Cons lst) → looks up and returns object associated with key in association list (lst)

- `F(assoc 'b '((a 1) (b 2) (c 3))) = (b 2)`

Some: Exists, returns first value that satisfies predicate

- **Some is False (null) if the set is null**

Every: All, returns boolean

- **Every is True if the set is null**

Subset: returns Cons which matches the subset of the original Cons

Divide and Conquer

By cutting a problem in half each time, the solution is `O(log(n))` if you throw away one half each time (Binary Search of a sorted list) or `O(nlog(n))` if you process both halves (adding together all values in a tree, sorting, etc...).
Example: Merge Sort

Fair vs. unfair data structure
If there is a possibility that an element in the structure will never get accessed, that's unfair. If every element, given enough time, will be accessed, it's fair. This assumes that elements are potentially being added to the structure while it's being used.

Stack:

LIFO - Last In, First Out
Not fair structure.
push(), pop() are O(1)
Uses of Stacks:
Most programming languages keep variable values on a runtime stack
Web browsers keep stack of previously visited web pages, pops when back button used to traverse a tree (searching for something) is hit, XML+
E.g. Input: {A, B, C, D, E} Output: {E, D, C, B, A}
Another E.g `push(A) push(B) pop() push(C) pop() pop()` -> B C A

Queue:

FIFO - First In, First out
Fair data structure: An entry will eventually be removed.
Uses of queues:
Processes in operating systems that are ready to execute, jobs to be printed, packets ready to be transmitted over a network.
E.g. Input: {A, B, C, D, E} Output: {A, B, C, D ,E}
Another E.g `insert(A) insert(B) remove() insert(C) remove() remove()` -> A B C **{FIFO IS ALWAYS THE SAME ORDER}**

Trees:

A form of a graph composed of nodes and links.
Link
- a directed pointer from one node to another. Each node (except root) has one incoming link from its parent. All nodes are reachable from the root.

Root -
- no incoming links. The top node of a tree, from which all other nodes can be reached.

Children -
- The nodes a particular node links to. If a node has no children, it is called a leaf.

Interior node - A node with children.
Leaf node - *Typically has parents**, no children
*A root node can be a leaf node, apparently.

Things that can be expressed as trees:
Arithmetic expressions
Computer programs.
Sentences in a language.
Big O for trees:
b - breadth or branching factor is the (average) number of branches per interior node
Special case: Binary tree, b = 2 (not average)

Example: Chess has a branching factor 35 because after every move there is a possible 35 different moves to make. Very high branching factor.

d - depth, is the height of the tree, $\log_b(n)$

n - number of leaf nodes. $\mathbf{n} = b^d$

Tree algorithms will typically be:

`O(log(n))` if one half is abandoned at each step. (e.g. binary .search)

`O(nlog(n))` if all branches are processed. (e.g. sort)

Depth-first search:

Recursion will go deep into the tree before it goes across. Search will quit and return an answer when a node is a terminal failure node or a goal node. DFS takes `O(log(n))` space - same as pre order ← Depth first search can use any of the three types of orders

Balanced binary trees:

A tree in which the heights of the subtrees are approximately equal

These include: AVL, Red-Black, and Splay Trees.

AVL Tree:

- Heights of right and left subtrees differ by at most 1.
- Advantage: Approx. **`O(log(n))`** search and insert time.
- Disadvantage: Complex code. Edge cases. A lot of memory (2 pointers)

B-Tree:

- Used when a tree is too big to be kept on memory, it is kept on a disk.
- Because it is being stored on a disk, you want to minimize the number of disk accesses because they take too long, to do this, **the branching factor should be very large.**
- Desired record found at a shallow depth
- Since a node can have a wide range of children, $\frac{m}{2}$ to m, an insert or delete will rarely go outside this range. It is rarely necessary to rebalance the tree.
- Insertion is typically **O(blocksize)**
  - **Question: What is blocksize referring to?** one interpretation: B tree is composed of key - pointer - key - pointer. My assumption is that this structure is a linked list, not a hash table. So these keys have to be traversed in order to find out where to insert data. At worst, data is inserted through the pointer at the end of the B-tree key-pointer "block". (unless it's referring to the complexity of the B tree being expanded into new blocks bc of overflow)
- Should only be used if the tree is **too large to store in main memory.**
-

Red-Black:

- Nodes are colored red or black, the longest path from the root to a leaf is no more than twice the length of the shortest path.
- A **red–black tree** is a kind of self-balancing binary search tree. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

Splay tree:

- Rebalanced so that recently accessed elements are near the top of the tree and can be accessed quickly the next time.

c

Quadtrees:

- Each (interior) node has 4 descendants (as opposed to binary trees which have 2 descendants)
- Can be represented as a planar graph, with a node representing one of the four quadrants
- Can be used for images (e.g. if one node's descendants has only one color -> just color the parent node -> save space)

Gedanken tree -

- a conceptual tree; for example, an array used to store tree using an array).
- An array is not literally tree-structured, but you can use its stored data in a way that pretends it's a tree.

Pattern Matching:

- uses: physics, algebra, calculus, translation of languages, optimization
- subst: subst(new,old,tree) replaces with new object onto old string in specified tree
  Example: (subst 'blue 'green '(green eggs and ham)) → (blue eggs and ham)

- sublis: substitute into a list with bindings in a list.
  Example: (sublis '((?man adam)(?woman eve)) ' (?man loves ?woman)) →
  (adam loves eve)

- equal: Checks for tree equality by seeing if structures of tree are same and if leaf nodes are equal.
  Example: (equal '(a)'(a)) → true

- match: (match(- ?x ?x) '(- z z)) -> binding list((?x z) (T T)) question: why is there an extra set of parentheses? <--Answer to question. Because a binding list is a list of bindings so for instance if there were two bindings it would be ((?x z) (?y panda)). The outer parens represent the initial list.
  - (match(- ?x ?x) '(- z y)) -> null. **would match (- z z)**
    - Why is this null? --- in pattern matching, variables *must be consistent*: in this case, it is not
  - (match(- ?x ?y) '(- z y)) -> binding list((?x z)(?y y))

  From notes: (match '(go ?expletive yourself) '(go bleep yourself)) -> ((?EXPLETIVE BLEEP) (T T)) //where does (T T) come from?

I

Q: Someone please answer above, I am so confused about (T T).
A: Says in the page right below: The dummy binding(T T)is used to allow an empty binding list that is not null

(T T) is a dummy binding. All it does is ensure that the binding list that match returns isn't null.

- transform: (transform '((-?x ?x) 0) '(-(fx)(fx)))) → 0 m
  - finds the bindings that work using match (if match != null, use that), then substitutes using that bindings, and continues to use match until all match == null

Does anyone have a more thorough example/explanation of transform?
When you call on transform, it checks if the pattern and list matches on their OP node. If they match, it then checks LHS, it associates ?x in the pattern with the variable in the list. It then checks the RHS of the pattern, if RHS node is the same as LHS node then the list RHS must match the LHS also. If so, return the answer 0.

Hashing:

Convert a key to an integer so that it can be used as an array index.
Randomize, two different keys usually hash into different values.
Typically one way, when you hash something, you usually cannot return back from the hash
Two different keys hashing to the same value is called a collision.
    When many keys do this, it's called clustering.
.hashCode() returns an int
    Must always return the same value during a given execution of a program.
    If two objects are .equals(), .hashCode() must return the same value for both.
    Can be negative, but is rarely prime

load factor lambda **λ** is the ratio of hash table entries to table size. Ideally want it to be < 0.7. Higher the lambda = greater probability of a collision.

Rehash is a way to handle a collision by rehashing the key using a different hash function.
Extendible Hashing ct of entries for items with the same hash value. <--Not limited to linkedlist. More accurate answer would be a Collection of values
Can be used when a table is too large to fit in main memory and must be placed on disk.(like B-tree)

Hashing with Buckets (separate chaining): A.K.A. Extendible Hashing
Uses a linked list Separate chaining - each
bucket is independent, and has some sort of list of entries with the same index, used to resolve hash collision.
technically, hashing with buckets is O(n) (if it's asked on a test), but it actually acts as O(1)

Set Interface in java: does not contain duplicate .equals() elements

Treeset interface: maintains set in sorted order (logn time)

Map interface: associates keys with values

XOR

- Boolean - returns true if its 2 input values don't evaluate to the same truth value
- Uses: the main idea is that if you have say, $x \oplus y$, you can get x via $(x \oplus y) \oplus y$, and vice versa.
    - Cryptography - *text* ⊕ *code* looks like garbage, but ⊕ *code* will restore this (similar use for backgrounds)
    - Doubly linked lists: store pointers as *next* ⊕ *previous*
    - RAID (redundant array of independent disks) : a disc stores $a \oplus b$: if one fails, use the other as to get the failed disc
    - hashing, wireless networking (broadcast message as $x \oplus y$)

Binary Heaps: A heap is a **complete** binary tree, meaning all nodes are filled except for bottom-right hand part

- adding a value will add it to the bottom-right hand part, and the heap self-balances so that it will rearrange itself to be sorted
- a heap is a gedanken tree, meaning it is actually stored in an array but acts like a tree
- parent is always less than descendants in a min heap, and vice versa
- for array index to heap: parent i/2; left child: 2*i right child: 2*i+1 (if i is the current node)
- Useful for algorithms like Prim's and Dijkstra's because of their relation to Priority Queues

Priority Queues:
Priority queues differs from a regular queue because the priority queue is a **binary heap** rather than just a normal array

- the queue sorts and rearranges itself as elements are added/removed
- items have a "priority value" that is sorted in the binary heap; adding an item with a certain priority does not necessarily add to the end, but will be added so that the queue's priority is sorted
- insert and deleteMin are **O(logn)**, not O(1) for regular queues, since the heap has to always be sorted
- Used for Prim's and Dijkstra's algorithm

PriorityQueue in Java:

- add(item) is the method we called insert
- peek() returns but doesn't remove the min element
- poll() removes and returns the min element, the method we called deleteMin() above.

Sorting: ***http://www.sorting-algorithms.com/insertion-sort***

Just an interesting illustration of various algorithms and their properties^

stable - does not change relative position of items with equal keys
in-place - no extra storage required to perform the sort
on-line - can sort as it receives new information one at a time with relative ease

Insertion sort:

Insertion sort finds the first item in a list that is not in correct order and makes a hole there. It then moves the hole to the left until the hole is in the correct spot, then inserts the item into the hole. Rinse/repeat until entire list is sorted.

O(n^2) only suitable for small n's or almost sorted input.

If input is almost sorted, Insertion sort is O(n+d) where d is the number of inversions or pairs of items not in correct order.

- stable
- in-place
- on-line
- Visualization: https://en.wikipedia.org/wiki/File:Insertion-sort-example.gif

Heapsort:

- **O(n*log(n))**
- NOT stable
- in-place
- NOT on-line
- visualization: http://commons.wikimedia.org/wiki/File:Heapsort-example.gif#mediaviewer/File:Heapsort-example.gif

Merge Sort:

Merge sort breaks the data into two halves until you reach a base case (one item) then merges back together.

- **O(n*log(n))**
- stable
- NOT in-place
- NOT on-line (https://piazza.com/class/hzblglzimpi7az?cid=958)
- can be parallelized easily https://commons.wikimedia.org/wiki/File:Merge-sort-example-300px.gif
- has good memory locality and cache performance
- Visualization: https://commons.wikimedia.org/wiki/File:Merge-sort-example-300px.gif

Quicksort:

Picks a pivot, reorders so lesser objects are in front of fparallepivot, recursively sorting sub-lists of lesser and sub-lists of greater values.**typically `O(n*log(n))`** **worst case `O(n^2)`**
Important to choose a good pivot, using the median of the first, middle, and last
is a good chance at a good pivot, not guaranteed, therefore (in introsort):

Change to Insertion Sort when size becomes small (=< 20)
Change to Heapsort after a certain depth of recursion
Easily parallelized, each subarray can be sorted independently.

- NOT stable
- in-place
- NOT on-line
- Visualization: https://www.tutorialspoint.com/data_structures_algorithms/images/quick_sort_partition_animation.gif

Radix Sort:

Sorts input into bins based on the lowest digit; then combines bins in order and sorts on the next highest digit & so forth.

O(n * k) k is key length, approx log(n) but if there are many items for each key, radix is
more efficient than O(n*log(n))

The idea of radix sort is to sort the input into bins based on the lowest digit; then combine the bins in order and sort on the next-highest digit, and so forth.
Was invented before computers (IBM, census)
Bins can be stored on external storage, so memory is not an issue.

- Can be parallelized
- Stable
- NOT in-place
- NOT on-line
- Visualization: https://www.cs.usfca.edu/~galles/visualization/RadixSort.html

"We said that Insertion Sort was on-line. That is basically the only one that we have studied that is." -Novak
Side note - Merge sort is supposedly also on-line as seen
"can efficiently merge in updates to an existing sorted array" -Class notes, 233 ← no? ← i don't see it either

- **Merge sort is usually NOT on-line** https://piazza.com/class/hzblglzimpi7az?cid=958

uniqueness:
merge - not in-place
quick - not stable
insertion - is on-line

Algorithm	**Big O**	**Stable** (does not change the relative position of items with equal keys)	**In-place** (does not require any additional storage)	**On-line** (can sort items as it receives them one at a time)
Insertion	$O(n^2)$	Yes	Yes	Yes
Merge	O(nlogn)	Yes*	No	No
Quick	O(nlogn) OR $O(n^2)$	No	Yes	No
Heap	O(nlogn)	No	Yes	No
Radix	O(n*k)	Yes	No	No

*dependent on implementation, but in most implementations it is stable.
merge/radix is stable
quick/heap is in-place
insertion is all
Java sorting library

- *Collection interface or list class* - sort by converting to an array(if a list, it'll be converted back)

Graphs:
A graph has a set of nodes (vertices) and a set of edges that connect nodes.
A path is a sequence of nodes connected by edges/arcs.
Length of a path: the number of edges/arcs
Cycle - a path from a node to itself
Acyclic graph - A graph with no cycles
Adjacency list - A list of all nodes with edges to a given node

Examples of Graphs:

- Road network between various cities (distance between cities is cost of edge)

- Airline network.
- Communication network (Internet, etc..)
- Social networks (Facebook, myspace, etc...)

……………………………….

Sparsity and Density:

- Graphs are dense if most nodes have edges to most other nodes.(or when $|E| = O(|V|^2)$
- Graphs are sparse otherwise.
- ***Most real-world graphs are sparse*** unless they are small.

+

Representations

Adjacency Matrix

- boolean matrix (with a 1 at position (i,j) if there's a link connecting nodes i and j
- **symmetric if the graph is undirected**
- storage is $O(|V|^2)$- good for dense graphs (else, excessive storage)

Adjacency List

- each node has a list of *adjacent* nodes (usually a linked list)**directed, each link is represented twice**
- Storage is $O(|V| + |E|)$: good for sparse graphs

Implicit

- graphs that can't be represented explicitly (i.e. future chess moves)
- only a part of such a graph will be considered

Topological sort

- order in which things are done: in a Directed Acyclic Graph (DAG), it specifies the path from two arbitrary vertices, and isn't always unique

Minimum Spanning Tree (MST)

As put by Novak; The cheapest way to wire internet to a whole area.

- A subgraph (tree) of a given undirected graph, with all nodes and a subset of edges, such that all nodes are connected and the sum of each arcs is minimized
- MSTs aren't unique, but their costs will be the same

Dijkstra's:

- Used to find the **shortest path** from a **root to *every*** node in the graph.
- Looks for the cheapest node, if it finds a node that has already been visited, compares the current path and the established path.
- **greedy algorithm**: picks best choice, in this case cheapest node, for current situation

Prim's:

Similar to Dijkstra's, used to find the ***minimum spanning tree***, that is, the tree that includes all nodes with the **lowest *total cost of edges***.

(assuming the list is already sorted)

A*:

- Uses **heuristics** to find the shortest path ***from a root to a specific node***.
- Uses both actual distance to the current node and estimated cost to the goal node.
- Unlike greedy, heuristics will take care of local maximas and minimas
- Good heuristics make for a highly efficient search, poor heuristics may help, but are kept on track by the actual cost to the

current node.

- Much more efficient for finding shortest path to a node, because it only looks for one node, instead of the shortest path to all nodes.
- **If** the heuristic **does not overestimate**, A* will **find the same path** as Dijkstra to a given node **with less work**.

Functional Programming

- All operations are done by functions
- Functions cannot modify arguments and have NO *side-effects* (i.e. printing, setting, writing to disc)
- Adaptable to parallel programming
- If arguments are commutative and associative - then the order of arguments is not important

MapReduce

- Difficult to make one computer CPU faster, but have plenty of them
- Parallelism is difficult - synchronization and possible failures
- master/slave
- Ensures atomicity in file systems - workers will rename temp file name to a final name (unless it is in that form)
- mapreduce works on programs over large data
- map: input -- > intermediate val (as a set of key-val pairs of string format) - sorting on keys for grouping
- reduce: intermediate -- > output

Big O of a lot of random functions:
length(x) = it's `O(n)` for linked lists,`O(1)` for arrays) - arrays store their length for O(1) access
reverse(x) = `O(n)`
append(x, y) = $O(n_x)$ append makes a copy of the first input list, and reuses the second input list. Therefore, its Big O depends only on the size of the first input. The Big O is *O(n)* because the first input list is copied, one element at a time. destructive is Big O(1).
mapcar() = O(n)
member(x, y) = O(n)
intersection(x, y) = O(n*log(n)) because sort then merge. can be n if using hashset.
union (x,y) = O(n*log n)
setDifference(x, y) = O(nlogn). **sort first,** if sorted O(n).
merge = O(n) if sorted
sort = O(n*logn)
mergesort = O(n*logn)
Binary search = O(log(n))
assoc = O(n)
push/pop = O(1) (logn for priority queue)
Divide and conquer generally = O(logn) as in binary search (throwing away half of of the tree each step) or O(nlogn) as in sorting (processing both halves of the tree)
API stuff = at top, page 106 notes. Basically array list is random access, linked list is O(1) add/remove for front and end
QUESTION: What do Hashmap/Treemap/Treeset/Set/other collections do and how to implement?

Comparing ArrayList and LinkedList
If the list contains n elements, the Big-O of operations is
as follows:

Method	ArrayList	LinkedList
get, set ith	O(1)	O(n)
… at front or end	O(1)	O(1)
add, remove ith	O(n)	O(n)

… at front	O(n)	O(1)
… at end	O(1)	O(1)
… in ListIterator	O(n)	O(1)
contains	O(n)	O(n)

Arraylist: Advantages: get(i) and set(i) are O(1) for all positions
Disadvantages: add(i) and remove(i) are O(n) for all positions except at end
Sometimes you have to create new array to add more elements
Wasted space (uses more space than needed)

LinkedList (Doubly-Linked List): Advantages: get, add, set, and remove for front and end are O(1) - good for stacks and queues
Disadvantages: get, set, add, remove for random positions are O(n)
Can't write destructive functions to setRest (more steps for constructive)

leaf nodes in a tree = b^d, where b = breadth and d = depth n = b^d
copytree = O(n)
substitute = O(n)
sublis = O(n^2) ??? *<--- OP of this, someone asked on piazza and Novak said didn't matter, yay*
hashing/accessing a hashtable = O(1)
insertionsort = O(n^2) if random, O(n+d) when almost sorted, where d is the # of inversions/pairs not in order
mergesort= O(nlogn)
heapsort = O(nlogn)
quicksort = O(nlogn) mostly, worst case O(n^2) (usually is the fastest of them all) (depends on chosen pivot)
radix sort = O(n*k), where k is usually logn (usually only for numbers/bitcode),
mapcar = O(n) - applies function of first argument to 2nd argument

```
(mapcar 'square '(1 2 3 17)) -> (1 4 9 289)
```

mapcan = O(n) - concatenates results, empty list vanish
reduce = O(n)

Sample Questions

- Why is it important that a search tree be balanced?
  - Balanced trees give us fast searches: O(log n) vs. ~~O(n^2)~~ O(n) <- shouldn't it be O(n) if it is completely unbalanced? It would be just a regular linked list, which is O(n). Yes.
- What is the most significant difference between B-trees and balanced binary trees?
  - B-trees have a high branching factor(on disk rather than RAM) and binary trees have the lowest possible branching factor (2)
- What is the motivation for having special techniques to store sparse arrays?
  - To save memory
- Describe three kinds of problems that can be done using pattern matching.
  1. Algebra
  2. Differentiation (Calculus)
  3. Translating computer languages
  4. optimization
- Describe the rules for matching a pattern variable.

    - a variable will **match anything** but **it must do so consistently**
- Suppose that data items are stored in a HashMap and that the items need to be sorted. How would this be done using the Java libraries?
    - Get values and sort them using Collections.sort
- Suppose that we would like to sort a set of students, first by age in years, and within the same age, by GPA. How would this be done using the Java libraries?
    - Write a comparator (could someone please expand/explain this?--You would write a comparator similar to the one we wrote in asg4 to sort the Accounts first by name and then by amount). This is needed for any custom sorting.
    - public int compareTo(student s), returns negative if this student is less than s, 0 if this student is equal to s, positive if this student is greater than s
- How does the Big O of hashing compare with that of AVL trees?
    - Big O of hashing O(1), as long as the load factor is kept below .7.
    - Big O of AVL trees O(log (n)) (this is a balanced search tree, where the heights of any node's subtrees differ by at most 1)
- Describe how hashing with buckets simplifies the code required.
    - We don't have to worry about collisions or expanding the table
    - But if collide, use buckets (a linked list/array containing all values of same hashcode) or rehash using a bigger array
- Describe three uses of priority queues.
    1. Operating systems, scheduling processes
    2. Greedy algorithms (i.e. Dijkstra's, Prim's algorithm)
    3. Discrete event simulation

- What is the Big O of Insertion Sort? Is it stable? In-place? What would be an appropriate use of Insertion Sort, and why?
    - O(n^2)
    - Stable and in-place, on-line
    - works well if an array is **almost sorted**
    - **Use: Sort new customers onto existing sorted list. Why:** Because insertion sort can add new items one at a time without having to take down the entire network system: **on-line** (able to accept new items one at a time and process them efficiently)
    - Adding to an already sorted list
- Give an advantage and a disadvantage of Merge Sort of an array.
    - **Advantage:** it is reliably O(n * log (n)), easily parallelized(dividing into subproblems then combining), less number of comparisons needed
    - **Disadvantage:** not in-place(bc dividing), requires 2X as much storage (tidbit: big-O of garbage collection is complicated)
- Give examples of problems whose representations are graphs.
    - Airline routes
    - Your brain
    - The internet

    - Nervous System
    - Friends on Facebook
- Depth-first search of a tree is easily done using a recursive function. How is search of a graph different?
    - A graph doesn't have a starting node or bottom/end (like leaves in a tree) and it is easy to get into a cycle or loop so we must keep a data structure to keep us from getting in a loop (*visited* in Dijkstra's)
- How do the computer systems on which MapReduce runs differ from ordinary computers?
    - 1 arbitrary computer is selected to be a master and just runs the master program
    - Lots of plain old computers
- Describe how MapReduce provides fault tolerance.
    - A given file is stored in multiple places
    - a master monitors the workers
        - Pings the workers for status
        - Reassigns workload of a dead computer to other computers
    - optimize by trying to choose the machine closest to the data
        - better yet find a computer with the data on its disk

Big O:

- If f(x) = 3 x^2 + 4 x + 7, what is its Big O? O(n^2)
- What is an optimal Big O for sorting? O(nlog(n))
- Which is better, O(n^2) or O(nlog(n)) ? O(nlog(n))
- What is the Big O of the following code? O(n^2)

```
for ( int i = 0; i < n; i++ )
         for ( int j = 0; j < n; j++ )
                x[i][j] = 0;
```

Linked Lists: *What is the result of (rest '(a b c))* *(b c)*

Q: shouldn't this be null? A: no because it's inside the first set of parenthesis. It would be null if it was (rest ‘ ((a b c)( ))) the only time rest would be null is if a list contains no elements/1 element

- What is the Big O of reverse of a list of length n? O(n)
- What is the result of (mapcar 'twice '(3 4 5)) given (defun twice (x) (* 2 x)) (6 8 10)

Stack, Queue:

- If the following operations are done on a Stack, what comes out? Push a, Push b, Pop, Push c, Pop, Pop.
    - b, c, a
- If the following are operations done on a Queue, what comes out? Insert a, Insert b, Remove, Insert c, Remove, Remove.
    - a, b, c (british are boring- always in same order as inserted) <lol

Java Collections:

- If coll is a LinkedList of Integer, write a function to add up all the elements. (O(N))

```
int sum = 0;
for(Integer x : coll)
      sum += x;
```

Trees:

- Write the following expression as a tree: x = (a + b) * c

```
     =
   /   \
 x       *
       /   \
     +       c
   /   \
  a     b
```

- Give an advantage of the first-child / next-sibling form of tree.
  - It can save memory and have unlimited number of children (visualization of first-child / next-sibling form can be found here)
  - Unlimited children
- What is Big O of search for a specified String in a binary search tree containing n Strings, if the tree is balanced?
  - logn
- In a tree with branching factor b and depth d, how many leaf nodes are there? How much stack depth is required to search such a tree?
-
  - b^d leaf nodes; stack depth of $\log_b(n)$