COMPSCI 201 MIDTERM 2 REFERENCE SHEET

Arrays

- .asList(), .toArray(), List to Array with loop

Strings

- += concatenation O(n) because immutable
- StringBuilder.append() = O(1) per char; large capacity until overflow, geometric growth

ArrayList

- implements List with Array, by keeping an array with extra space at the end
    - Array out of space → make one twice as large and copies old over
- get() = O(1), contains() = O(n), size() = O(1)
- add() = O(1) at end (amortized bc copying), O(n) at beginning (has to shift over)
- remove() = O(N), have to shift all elements

HashSet

- unordered but efficient; internally creates a HashMap to store elements of set
- add(), remove(), contains(), size() = O(1)

HashMaps

- calculates hash (int) of a key with .hashCode() to see where to store values (diff keys can have same hash)
- puts it in a "bucket" of k,v pairs with that hash index
- put() adds to hash index bucket, updates if present = O(1)
- get() gets value at computed location to get hash bucket iterates over hashBucket keys to find .equals() key = O(1)
- containsKey() checks if location empty = O(1)

TreeSet/TreeMap

- ordered set/map, slightly less efficient than HashSet/HashMap
- keeps values (for Set) and keys (for Map) sorted by "natural ordering," uses special kind of binary tree
- most methods are runtime complexity O(log n)

Big O

- outer(inner(n)) → runtime of inner + runtime of outer on return value of inner

$O(2^N)$	Exponential	Calculate all subsets of a set
$O(N^3)$	Cubic	Multiply NxN matrices
$O(N^2)$	Quadratic	Loop over all *pairs* from N things
O(N log(N))	Nearly-linear	Sorting algorithms
O(N)	Linear	Loop over N things
O(log(N))	Logarithmic	Binary search a sorted list
O(1)	Constant	Addition, array access, etc.

LinkedList

- each node has value and reference to another node
    - next returns node, not value
- add() and remove() at front = O(1) → change pointers
    - removing first → save reference to second, set first node to null (break connection), set first to second
- get() = O(N), loops through each element until null
- contains() = O(N), loops through list until you find it
- traverse with a while loop (list != null)
- null pointer exc if you call method/variable on null node

recursion

- base case and recursive calls (get answer on input subset)
- method calls a clone with its own frame, doesn't call self
    - call stack of methods in a cue
- invariant: true statement while loop is running
- ex: reverse
    - iterative → temp = list.next list.next = rev rev = list list = temp (rev is reversed so far)
    - recursive (O(N)) → base case if list or list.next == null, store recursive call on list.next, set list.next.next ot list, set list.next to null, return stored recursive call

recursion cont'd

- recursive runtime
    - T(N) = a * T(g(N)) + f(N)
    - T(N): runtime of method with input size N
    - a: number of recursive calls
    - g(N): how much input size decreases on each recursion
    - f(N): runtime of non-recursive code on N

Recurrence	Algorithm	Solution
`T(n) = T(n/2) + O(1)`	binary search	`O(log n)`
`T(n) = T(n-1) + O(1)`	sequential search	`O(n)`
`T(n) = 2T(n/2) + O(1)`	tree traversal	`O(n)`
`T(n) = T(n/2) + O(n)`	qsort partition ,find k^{th}	`O(n)`
`T(n) = 2T(n/2) + O(n)`	mergesort, quicksort	`O(n log n)`
`T(n) = T(n-1) + O(n)`	selection or bubble sort	`O(n²)`

insertion sort without recursion → O(N2)

- loop through original list, maintain a new sorted list, insert one value at a time in order (looping through new sorted list)

mergesort

- base case → size 1, return list
- recursive case → mergesort(first half), mergesort(second half), return merged sorted halves with helper merge(first, second)
- merge() → add to a new list, traverse listA and listB with 2 indices with looping, compare the value at each of 2 indices and add the smaller
- O(N log(N)) → halves at each level so O(log(N)) levels, merge at each level with O(N)
- merge() → O(nA + nB) = linear

comparisons in mergesort

- one comparison per loop / per element merged
- O(CN log(N)), where C is complexity of compareTo() or compare() that may not be a constant value

Java API sort → Collections.sort() Arrays.sort()

- Timsort (like Mergesort) = O(N log(N))
- sorts in place, mutates input instead of making new

stable: does not reorder elements if not needed (ex: 2 elements are equal)

Comparable

- class objects that implement Comparable interface can be sorted → have a naturalOrder
- compareTo() → method is in class of this object type
    - < 0 if this comes before parameter, 0 if equal, > 0 if this comes after parameter
    - ^ if it's in the right order, return negative

Comparator

- use Comparator when not changing the object itself to compare non natural order
- Comparator c, c.compare(a,b)
    - < 0 if a comes before b, 0 if equal, > 0 if a comes after b
    - method is part of Comparator
- Collections.sort(*list*, Comparator.comparing(*ObjType*::*methodName*))
- to combine sequence of comparison → Comparator.comparing(_::_).thenComparing(_::_)
- with lambdas (for comparing something without a getter)
    - Comparator<*ObjType*> c = (*a,b*) -> (*what c.compare(a,b) should return*)
    - ex: (a,b) -> (a[0] - b[0]) for first elements
    - can use sort(*list, createdComparator*)

binary search → needs sorted list

- O(log(N)) → cut down search space by half at each step
- low (initially 0) and high (initially N-1) mark the limits of active search space
- loop while (low = high) → while there's anything left to search
    - loop invariant → if target is in array/list, it is in [low, high]
- compare mid ((low+high)/2) to target
- search in lower (high = mid-1) or upper half (low = mid+1)
    - not high = mid or low = mid to prevent infinite loop in edge cases
- does not guarantee the first or last index of a match if there are multiple