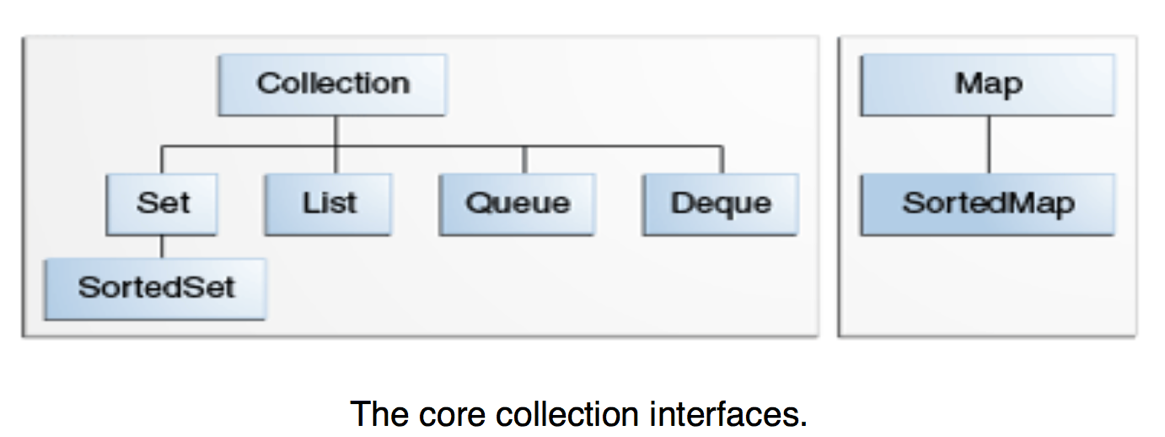
**VI. Data Structure**

1. **Collection Interface**



* **Collection Interface Hierarchy**
* **Collection** — the root of the collection hierarchy
* **Set** — a collection that cannot contain duplicate elements.
* **List** — an ordered collection
* **Queue** — a collection used to hold multiple elements ordered in FIFO
* **Deque** — same, but used both as FIFO and LIFO
* **Heap** --- priority heap
* **Map**: an object that maps keys to values, no duplicate keys
* **Set**
* **HashSet**
* initialCapacity: When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed
* add(E)
* clear()
* clone(): shallow copy, but can modify the cloned object without modifying the original object.
* contains(Object)
* iterator(): returns an iterator over the elements in this set
* remove(Object)
* **TreeSet**
* implement using a tree structure(red-black)
* the elements are sorted
* add(), remove(), contains() has time complexity of O(log (n))
* **LinkedList**
* implements with a doubly-linked list.
* get(): O(n/2)
* add()/remove()/element(): O(1); throws Exception
* contains()
* indexOf(Object)
* listIterator(start index)
* pop(): stack
* push(): stack
* set(index, E)
* isEmpty()
* toArray()
* **ArrayList**
* implements with a dynamically re-sizing array.
* ArrayList VS Vector

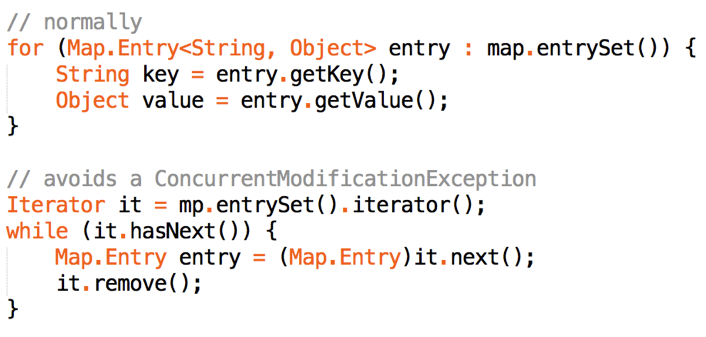
\* Vectors are synchronized; Array Lists are not.

\* Data Growth Methods

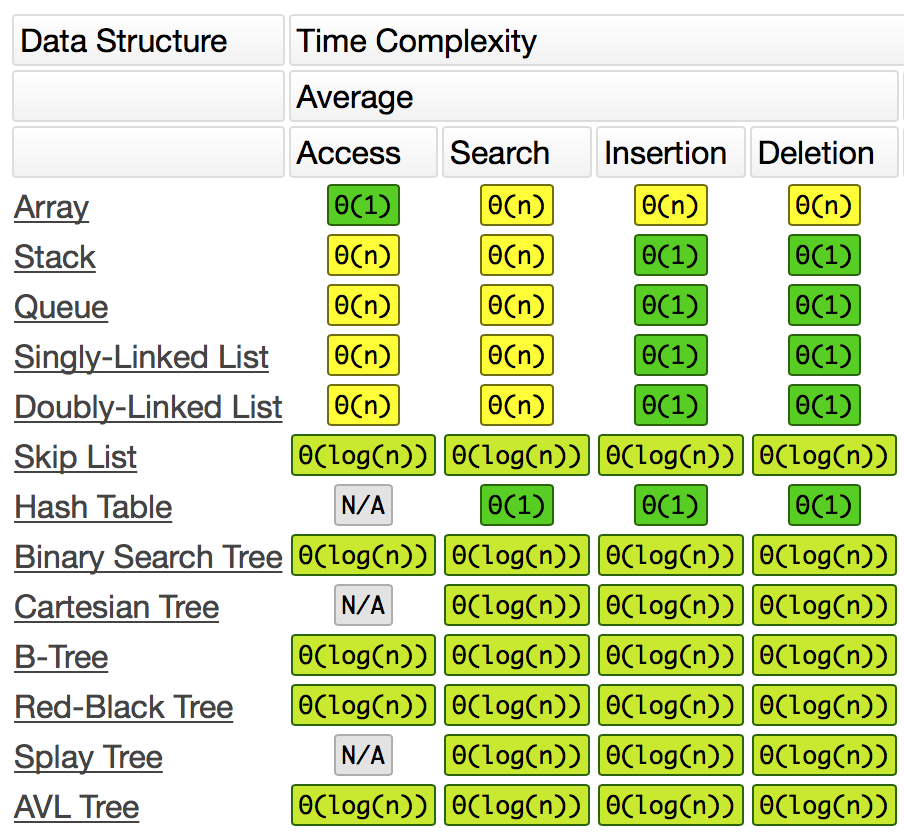
* get(): O(1)
* add()/remove(): O(n)
* isEmpty()
* iterator()
* contains()
* sort(Comparater<? super E>)
* subList(fromIndex, toIndex)
* toArray()
* **Queue**
* LinkedList (FIFO)
* offer()/poll()/peek(): return true/null/null value
* **Deque**
* Deque implements both stacks and queues
* ArrayDeque
* Deque<> dq = new ArrayDeque<>()
* addFirst()/addLast()/removeFirst()/removeLast()
* offerFirst()/offerLast()/pollFirst()/pollLast(): thread safe
* peekFirst()/peekLast()Stack
* **Map**
* HashMap
* clear()
* clone()
* containsKey(Object)
* containsValue(Object)
* keyset(): return set of key
* get(Object)
* **isEmpty(**)
* put(K, V)
* remove(key)
* values(): Returns a Collection view of values in this map

map.values().stream().filter(node -> node.val > 0).collect(Collectors.toList())

* Set<Map.Entry<[K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)>> entrySet()



* getOrDefault([Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) key, [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) defaultValue)
* putIfAbsent([K](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) key, [V](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) value)
* **Heap**
* PriorityQueue
* PriorityQueue()/PriorityQueue(int initialCapacity, Comparator<? super [E](http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html)> comparator)
* add()/remove(): O(logn)/O(1)
* contains(Object)
* iterator()
* offer()/peek()/poll()
* toArray()
* **Complexities of various data structure**



Heap peek: O(1) offer: O(log n) poll Max: O(log n)

* **Create Own Object Ordering**
* implements Comparable (set default Comparator for a class)
* public class Name implements Comparable<Name> {

@Override

public int compareTo(Name n) { … }

}

* implement Comparator
* new Comparator<XXX>() {

@Override

public int compare(XXX e1, XXX e2) {}

}

* Use Comparable if you want to define a natural ordering behavior
* Use Comparator if you want to define an external controllable ordering behavior
* Collections.sort(List, Comparator<>)
* Arrays.sort(T[], Comparator<>)

1. **Array**

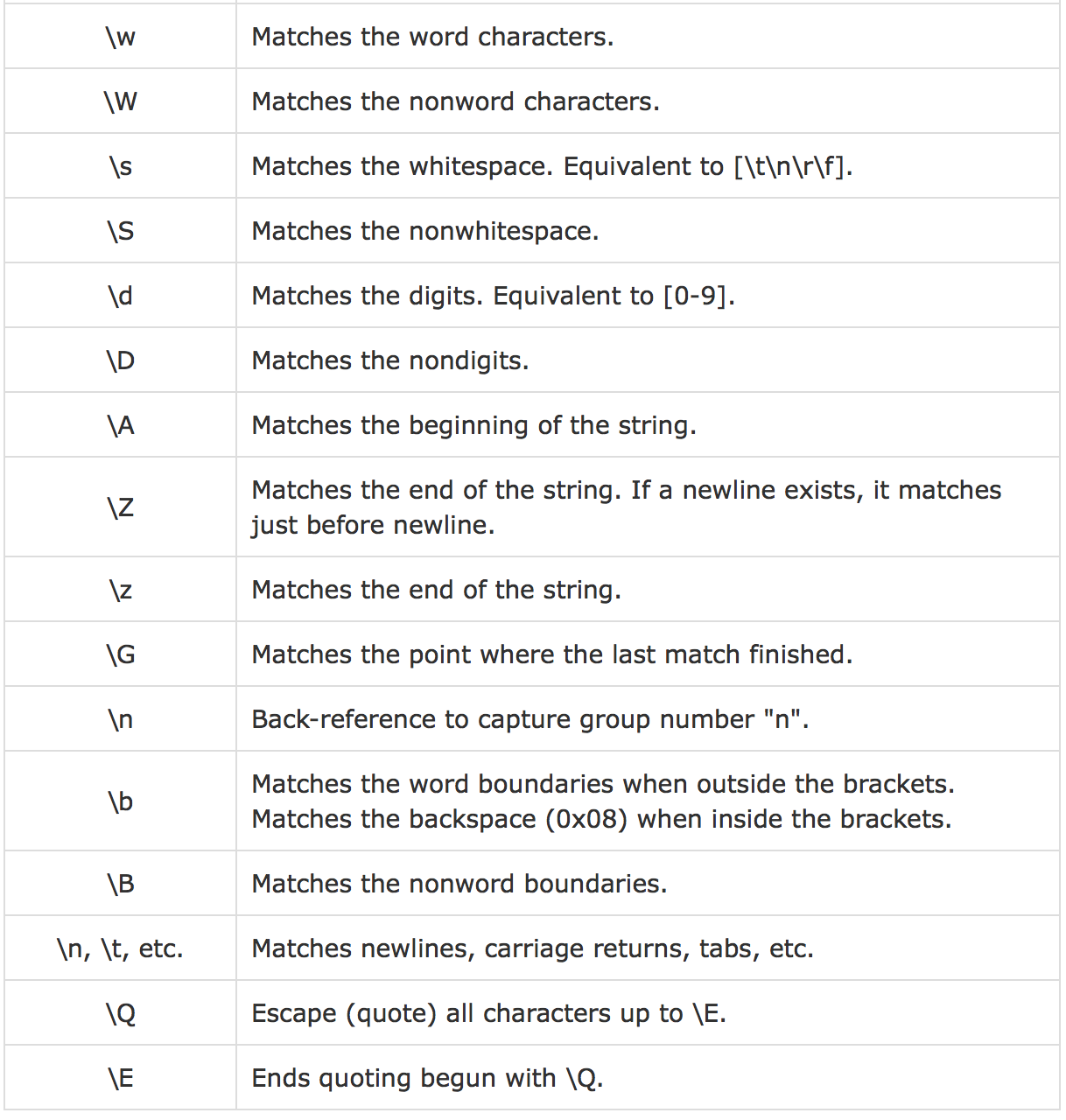
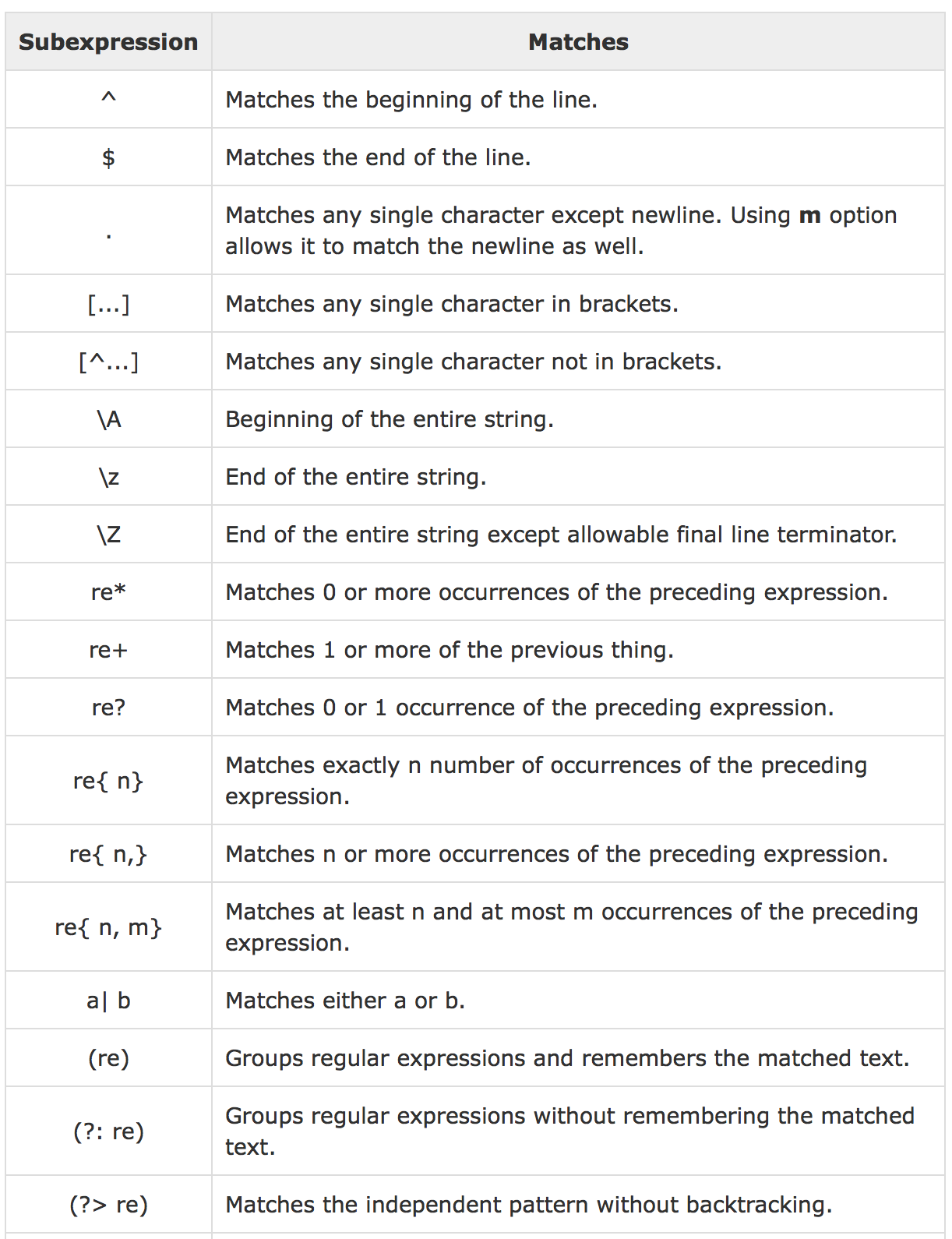
* **Problem1: Pair**
* For example: 2sum, 2remainder
* Sorted array? 2 pointer need sorted array
* Repeat element? Repetition need HashMap(cnt) rather than HashSet
* Return count or all indexes?
* Can I modify array?
* Utilize monotonically decreasing attribute in index difference problem
* **Problem2: Top K**
* Space limit? If memory not enough, cannot sort/use Hashing (need external sort)
* Min/Max problem usually can solve with heap
* Do we need to sort? If not, we can use partition with O(n) complexity

(See 3. Algorithm)

* **Problem3: Kth number/Meidan of 2 Array**
* O(log(m+n)) Merge partition (divide and conquer)
* (See 3. Algorithm)
* **Problem4: Rotated Array**
* The 1st key is to judge which side (left/right) is sorted?
* If (A[left] <= A[mid])) //means left side is sorted
* The 2nd key is to check if exist repeat element?
* {111, 1,0,0,1} or {1,0,0,1,1,1,1}
* If so, if (A[mid] == A[left]==A[right]) left++; //cannot judge sorted side

1. **String**

* **Regex expression**

****

* **Decode**
* ASCII: length=256
* UTF-8: 8bit
* [a-z]: length=32
* **Problem1: Extract Words From Sentence (white space removed)**
* Use HashMap<String, ArrayList<String>> memoriezed for DP
* If (contains(String input )) return memoriezed.get(input);

If (dictionary.contains(input)) res.add(input); //the all string is a word

For (int i = 1; i < len; i++) {

Prefiex = input.substring(0, i);

If (dictionary.contains(prefix))

For (String segSuffix: //recursive call with substring(I, len))

If (segSuffix != null)

Res.add(prefix + “ ” + segSuffix)

}

memorized.put(input, res);

return res;

* **Problem2: Count Max Words from String (white space removed)**
* For count number rather than return res, always possible to use DP
* Here use dp[i] to represent 0-i maximum number of words
* For () loop //record dp[] from substring 0-i with dictionary//initialize
* For (int I = 0; I < s.len – 1; i++)

For (int j = i+1; j < s.len; j++)

If (dp[i] > 0)

If(s.substring(i+I, j+1) in dictionary)

Dp[j]=Math.max(dp[j], dp[i] + 1);

Return dp[s.len - 1];

* **Problem3: Number Validation**
* Check null / “”
* Str.trim()
* Str.toLowerCase() //for easier process
* Check if invalid character exists
* +/- only be first or behind ‘e’; cannot be last character
* ‘.’ Cannot be only char or without other number or behind e
* ‘e’ cannot be first or last char; no number ahead; e exists before
* use Boolean flags to represent existence of special characters
* **Problem4: String to Integer**
* Invalid character exists?
* Data overflow? How to deal?
* Get possible sign character (may not have)
* While (p < str.len)

If (overflow) { process }

Num = 10\*num + (ch – ‘0’);

P++;

Return (!isNeg) ? num: -num;

* When will overflow?
* An N-bit two's-complement numeral system can represent every integer in the range −(2^(N − 1)) to +(2^(N − 1) − 1)
* [https://en.wikipedia.org/wiki/Two%27s\_complement - Addition](https://en.wikipedia.org/wiki/Two%27s_complement#Addition)
* How to deal with overflow?
* Return Math.MAX\_VALUE/Math.MIN\_VALUE instead
* Throw new Exception(…)
* Use Long type or BigInteger bi = new BigInteger(str);
* Static BigInteger valueof(long)
* BigInteger add/subtract/multiply/divde/(BigInteger val)
* Int compareTo(BigInteger val)
* **Attention**
* 2 String problem, lengths could be different, use if else to quick return
* use res.add(str.substring(0, str.len - 1)) to add without ‘unwanted’ character
* the first char of number cannot be ‘0’

1. **Lists, Stack and Queue**

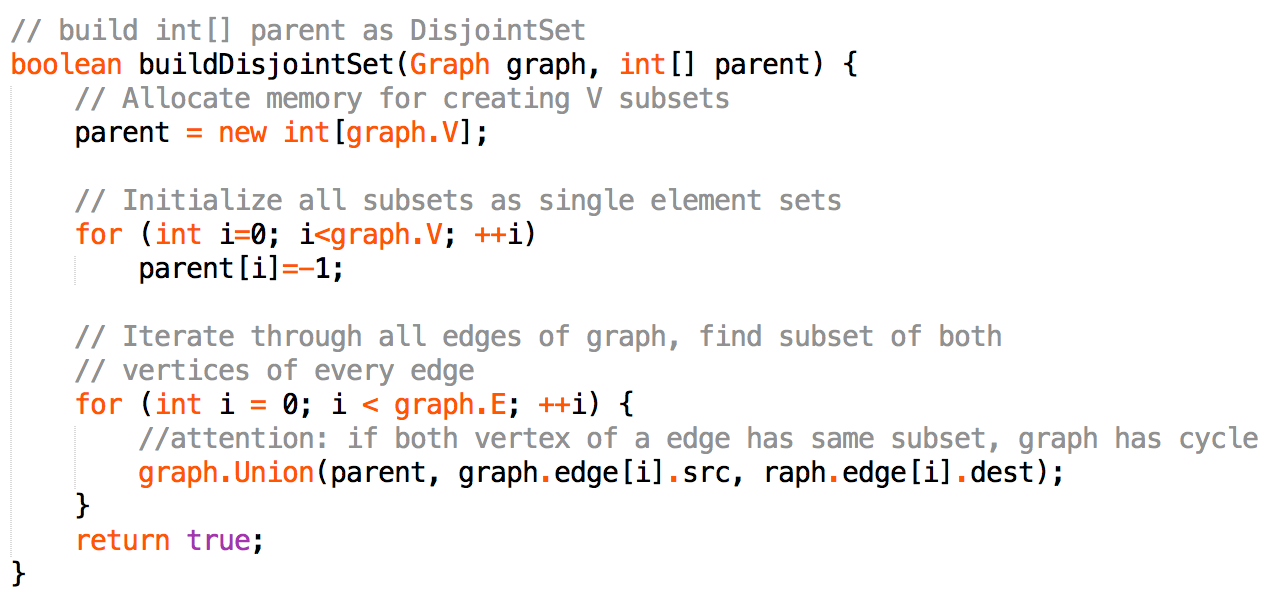
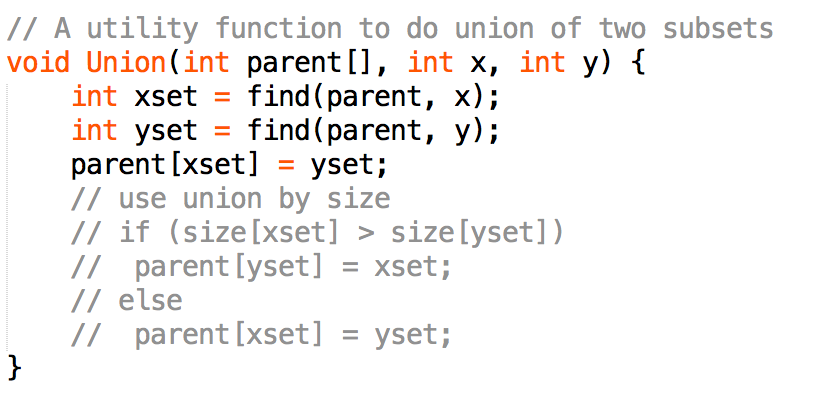
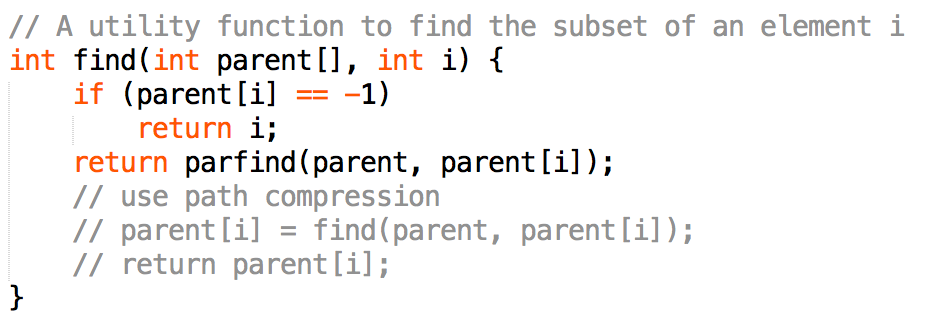
* **Solution1: 2 Pointers**
* Combine 2 lists
* Judge Directed Acyclic Graph & Find Cycle start node (fast and slow pointer)
* **Solution2: Recursion**
* Recursive Reverse Linked list(backtracking)
* Non-recursive Non-recursive solution
* Use ‘dummy’, ‘cur’, ‘next’ node
* Set head.next = null, and travel from second node
* **Worth Mention:**
* Use dummy node for easy return when final head is uncertain
* When comes to 2 linked list, their length may not the same

1. **Dictionary**

* ADT (Abstract Data Type) that maps keys to values
* Choice 1: Hash
* Each vocabulary term is hashed to an integer
* Advantage: Lookup is faster than for a tree: O(1)
* Disadvantage: No easy way to find minor variants
* Disadvantage: No prefix search
* expensive operation of rehashing
* Choice 2: B-Tree
* Advantage: Solves the prefix problem
* Disadvantage: O(log M) [and this requires balanced tree]

1. **Disjoint Set**

* **Union & Find**
* **Take O(n) time**
* **With Smart union / path compression(during find()), take O(logn)**

****

1. **Priority Queue (Heaps)**

* **Binary heap**
* Complete binary tree
* FindMin in O(1)
* Insert & Remove in O(logn)
* Java PriorityQueue: add()/remove()/element()
* Java implementation(Generic Class & T[] array)

