```
int binarySearch(A, key, n)
	begin = 0
	end = n-1
	while begin < end do:
		mid = begin + (end-begin)/2
		if key <= A[mid] then
			end = miid
		else begin = mid + 1
	return (A[begin] == key)? begin: -1
FindPeak(A,n)
	if A[n/2 + 1] > A[n/2] then
		FindPeak(A[(n/2)+1...n], n/2)
	else if A[n/2 - 1] > A[n/2] then
		FindPeak(A[0...(n/2)-1]), n/2)
	else A[n/2] is a peak
		return n/2
BubbleSort(A,n)
	repeat n times:
		for j <- 1 to n-1
			if A[j] > A[j+1] then swap(A[j], A[j+1])
SelectionSort(A,n)
	for j <- 1 to n-1:
		find minimum elem A[k] in A[j...n]
		swap(A[j], A[k])
InsertionSort(A,n)
	for j <- 2 to n
		key <- A[j]
		//Insert key into the sorted array A[1...j-1]
		i <- j-1
		while (i > 0) and (A[i] > key)
			A[i+1] <- A[i]
			i <- i - 1 //going R to L ensures stability
		A[i+1] <- key
MergeSort(A,n)
	if (n==1) then return //base case
	else:  //recursive conquer step
		X <- MergeSort(A[1...n/2], n/2)
		Y <- MergeSort(A[n/2+1...n], n/2)
	return Merge(X,Y,n/2) //combine soln
MergeSort(A, begin, end, tempArray)
	if (begin = end) then return
	else:
		mid = begin + (end-begin)/2
		MergeSort(A, begin,mid, tempArray)
		MergeSort(A, mid+1, end, tempArray)
	Merge(A[begin...mid], A[mid+1...end], tempArray)
	Copy(tempArray, A, begin, end)
MergeSort(A,B, begin,end)
	if (begin=end) then return
	else:
		mid = begin + (end-begin)/2
		MergeSort(B,A,begin,mid)
		MergeSort(B,A,mid+1,end)
	Merge(A,B,begin,mid,end)
QuickSort(A[1...n], n)
	if (n == 1) then return
	else
		p = Partition(A[1...n], n)
		x = QuickSort(A[1...p-1],p-1)
		y = QuickSort(A[p+1...n], n-p)
Partition(A[n...1],n,pIndex)
	pivot = A[pIndex]
	swap(A[1], A[pIndex])
	low = 2
	high = n+1 //due to invariant
	while (low < high)
		while (A[low] < pivot) and (low < high) do low++
		while (A[high] > pivot) and (low < high) do high--
		if (low < high) then swap (A[low], A[high])
	swap(A[1], A[low-1])
	return low-1
QuickSort(A[1...n], n)
	if (n == 1) then return
	else
		pIndex = random(1,n) //Choose pivot index pIndex
		p = 3WayPartition(A[1...n], n, pIndex)
		x = QuickSort(A[1...p-1],p-1)
		y = QuickSort(A[p+1...n], n-p)
```

```
END GOAL (3 way partioning): [<= x][x,..,x][>= x]
Option 1) 2 pass partioning
- Regular partioning
- Pack duplicates (using swaps)
Option 2) Standard Implementation
- Standard soln
- Maintain 4 regions of the array
-> [<= x][PIVOT][IN PROGRESS][>= x]
if (A[curr] < pivot) {
	Increment low
	Swap(A[curr], A[low])
	Increment curr
} else if (A[curr] = pivot) {
	Increment curr
} else if (A[curr] > pivot) {
	Swap(A[curr], A[high])
	Decrement high
}
QuickSelect(A[1...n],n,k)

	if (n==1) then return A[1]
	else Choose random pivot index pIndex
		p = partition(A[1...n],n,pIndex)
		if (k == p) then return A[p]
		else if (k < p) then
			return QuickSelect(A[1...p-1],k)
		else if (k > p) then
			return QuickSelect(A[p+1...n],k-p)
public TreeNode successor() {
	if (rightTree != null) { //base case
		return rightTree.searchMin() //searchMin juz recurses left all the way
	}
	TreeNode parent = parentTree;
	TreeNode child = this;
	while ((parent!=null) && (child == parent.rightTree)) {
		child = parent;
		parent = child.parentTree;
	}
	return parent; //note: if no successor, will return null
}

delete(key)
	TreeNode v = search(key)
	if (v.children = 0) //case 1: v has no children
		remove v
	else if (v.children = 1) //case 2: v has 1 child
		remove v
		connect child to parent
	else if (v.children = 2) //case 3: v has 2 children
		x = successor(v)
		delete(x)
		remove(v)
		connect x to left(v), right(v), parent(v)
//Balanced tree
right-rotate(v) //vice versa for left-rotate
	w = v.left
	w.parent = v.parent
	v.parent = w
	v.left = w.right
	w.right = v
insert(x)
	if (x < key)
		left.insert(x)
	else right.insert(x)
	height = max(left.height + right.height) + 1
if v is unbalanced & left heavy
	if (v.left is balanced || v.left is left heavy) then
		right-rotate(v)
	else if (v.left is right-heavy) then
		left-rotate(v.left)
		right-rotate(v.right)
delete(x)
1) if v has 2 children, swap it with its successor
2) Delete node v from binary tree (& reconnect children)
3) For every ancestor of deleted node:
	Check if its height balanced
	If not perform a rotation //May take up to O(logn) rotations
	Continue to root
//Order Statistics
Select(k)
	rank = left.weight + 1
	if (k == rank) then
		return val
```

```
//Order Statistics
Select(k)
        rank = left.weight + 1
        if (k == rank) then
                return val
        else if (k < rank) then
                return left.select(k)
        else if (k > rank) then
                return right.select(k-rank)
rank(node)
        rank = node.left.weight + 1
        while (node!= null) do
                if node is leftchild then
                        do nothing
                else if node is rightchild then
                        rank += node.parent.left.weight + 1 //adding rank of parent
                node = parentnode
        return rank
//Interval Search
intervalSearch(x) //O(n)
        c = root;
        while (c != null and x is not in c.interval) do
                if (c.left == null) then
                        c = c.right
                else if (x > c.left.max) then
                        c = c.right
                else c = c.left
        return c.interval
/*Orthogonal Range Searching
Algorithm: Query (find number of points in range)
- Find split node v (node withing range)
- Do left traversal on v.left
- Do right traversal on v.right */
FindSplit(low,high)
        v = root
        done = false
        while (!done)
                if (high <= v.key) then (v = v.left) //v.key too high
                else if (low > v.key) then (v = v.right) //v.key too low
                else (done = true)
        return v
RightTraversal(v, low, high) //key value increase
        if (v.key <= high) //key still lower than upp bound?
                all-leaf-traversal(v.left)
                RightTraversal(v.right, low, high)
        else  //if not check if left child within upp bound
                RightTraversal(v.left, low, high)
LeftTraversal(v, low, high) //key value decrease
        if (low <= v.key)  //key still above lower bound?
                all-leaf-traversal(v.right)
                leftTraversal(v.left, low, high)
        else //if not check if right child above lower bound
                leftTraversal(v.right, low, high)
```

- Big O notation
  $T(n) = O(f(n))$ → $\exists$ constant $c > 0$
  → $\exists$ ~~…~~ " $n_0 > 0$
  s.t. $T(n) \leq cf(n)$ $\forall n > n_0$

- Methods to solve recurrence
  1) ~~Continuous~~ Repeated Substitution
  2) Draw Tree
  3) Master Thm / Akra Bazzi
  4) Guess & inductive proof

Simple Recurrences
- $T(n) = 2T(\frac{n}{2}) + n \Rightarrow O(n\log n)$
- $T(n) = 2T(\frac{n}{2}) + 1 \Rightarrow O(n)$
- $T(n) = T(\frac{n}{2}) + n \Rightarrow O(n)$
- $T(n) = T(\frac{n}{2}) + 1 \Rightarrow O(\log n)$
- $T(n) = 1 + T(n-1) + T(n-2) \Rightarrow O(2^n)$

Master Thm: $T(n) = aT(\frac{n}{b}) + f(n)$ where $a \geq 1, b > 1$
- $f(n) = \Theta(n^d)$, where $d \geq 0$
① $a < b^d \longrightarrow T(n) = \Theta(n^d)$
② $a = b^d \longrightarrow T(n) = \Theta(n^d \log n)$
③ $a > b^d \longrightarrow T(n) = \Theta(n^{\log_b a})$

Sterling's approximation: $\log(n!) \approx O(n\log n)$

Order of size
$f(n)$
5 (constant)
$\log\log n$ (Double Log)
$\log n$
$\log^2 n$ (polylogarithmic)
n
$n\log n$ (log-linear)
n^3 (polynomial)
$n^3\log n$
n^4
2^n (exponential)
2^{2n}
$n!$

Hashing

Simple Uniform Hashing assumption
- Every key equally likely to map to every bucket
- Keys mapped independently

Assume: n items, m buckets

Sorting Algorithms

Name	Best Case	Avg Case	Worst Case	Extra Memory	Stable
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	No
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	Yes
Quick Sort	O(nlogn)	O(nlogn)	O(nlogn	O(1)	No

Hashing
- No successor/predecessor query
- Load $\alpha = n/m$ = E(items per bucket) //[n items, m buckets]
- Simple Uniform Hashing Assumption
1) n items, m = Ω(n) buckets
- E(search time) = 1 + n/m = O(n) (if m = be, Worst case: O(n)
- Worst case (insertion): O(1)
- Inserting n items, Expected max cost: O(logn)

Good Java Hash Function
- Defined hashCode(), Override equals()
- objects that are equals (including itself) must return same hash code
1) Enum all possible buckets
2) Uniform Hashing Assumption

1) Chaining (m buckets, n size of linked list)
- insert(key, val): O(1 + cost(h))
- search(key): O(n + cost(h))

2) Open Addressing (eg. Linear Probing/Weird Probing)
- Probe a sequence of buckets until you find empty bucket
- h(key, i): U -> {1…m} //i is number of collisions
- COST of operation (given uniform hashing): <= $\frac{1}{1-\alpha}$
- Prob: Operation cost degrades badly as α approaches 1