Union find:

```
//Initialize ~n
id = new int[N];
for (int i = 0; i < N; i++) id[i] = i;
//find(int i) O(log n)
while (i != id[i]) {
   id[i] = id[id[i]];
   i = id[i];
}
return i;
//union(int p,int q) O(log n)
int i = find(p);
int j = find(q);
if (i == j) return;
if (sz[i] < sz[j]) {
   id[i] = j; sz[j] += sz[i];
}
else {
   id[j] = i; sz[i] += sz[j];
}
```

Analysis of algorithm:
Big Theta Θ: Asymptotic order of growth. e.g. $5n^2+3n \log n = \Theta(n^2)$
Big Oh O(): Equal to or smaller than Θ.
Big Omega Ω: Equal to or bigger than Θ.
Tilde ~: Provides leading term.
e.g. $10n^2 + 5n \sim 10n^2$

Memory usage:
Object reference: 8 bytes
Array: 24 bytes +
Object: 16 bytes +
Padding: round up to multiple of 8B

Binary search:

```
int left = -1;
int right = -1;
int lo = low;
int hi = high;
while( lo <= hi ) {
 int mid = lo + (hi-lo)/2;
 if( array[mid] < target ) {
   lo = mid+1;
 } else if( array[mid] > target ) {
   hi = mid-1;
 }else if(mid == lo || array[mid-1] < target) {
   left = mid;
   break;
 } else {
    hi = mid-1;
 }
}
if( left == -1 )
 return new int[]{-1,-1};
lo = left;
hi = high;
while( lo <= hi ) {
 int mid = lo + (hi-lo)/2;
 if( array[mid] > target ) {
   hi = mid-1;
 }else if( mid == hi || array[mid+1] > target) {
   right = mid;
   break;
 } else {
   lo = mid+1;
 }
}
return new int[]{left, right};
```

Stack:

```
//Stack with linked list
//Node
String item;
Node next;
//isEmpty()
return first == null;
//push(String item)
Node oldFirst = first;
first = new Node();
first.item = item;
first.next = oldFirst;
//pop()
String item = first.item;
first = first.next;
return item;

//Stack with re-sizing array
//initialize(int capacity)
s = new String[capacity];
n = 0;
//isEmpty()
return n == 0;
//push(String item)
if(n == s.length) resize(2 * s.length);
s[n++] = item;
//pop()
String item = s[--n];
s[n] = null;
if (N > 0 && N == s.length/4) resize(s.length/2);
return item;//Avoid loitering
//resize(int capacity)
String[]copy = new String[capacity];
for (int i = 0; i < N; i++)
   copy[i] = s[i];
s = copy;
```

Queue:

```
//Queue with linked list
//Node
String item;
Node next;
//isEmpty()
return first == null;
//push(String item)
Node oldLast = last;
last = new Node();
last.item = item;
last.next = null;
if(isEmpty()) first = last;
else oldLast.next = last;
//pop()
String item = first.item;
first = first.next;
if(isEmpty()) last = null;
return item;
```

Elementary sorts:
Selection Sort (I/NS)
Time: $\sim n^2/2$ Space: O(1)

```
for( int i = 0; i < array.length; ++ i ) {
   int min = i;
   for( int j = i+1; j < array.length; ++ j ) {
      int compare = array[min].compareTo
(array[j]);
      if( compare > 0 )
         min = j;
   }
   exchange(array, i, min);
}
```

Insertion Sort: (I/S)
Time: $\sim n^2/4$ Space: O(1)
Best: ~n Worst: $\sim n^2/2$

```
for(int i = 1; i < array.length;++ i ) {
 for(int j = i;j > 0 && array[j] <= array[j-1];-- j)
   exchange(array, j-1, j);
}
```

Shell Sort: (I/NS)
Time: $\sim cn^{3/2}$ Space: O(1)
Best:~n $\log_3 n$

```
int N = a.length;
int h = 1;
while (h < N/3) h = 3*h + 1;
while (h >= 1) {
   // h-sort the array.
   for (int i = h; i < N; i++) {
      //insertion sort
      for (int j = i; j >= h && less(a[j], a[j-h]); j -=
h)
         exch( a, j, j-h);
   }
   h = h/3;
}
```

Merge Sort: (NI/S)
Time: ~n log n Space: O(n)
Best: ~n log n/2 Worst: ~n log n

```
//Merge:
System.arraycopy(array, low, aux, low,
high-low+1);
int i, j1, j2;
for( i = low, j1=low, j2 = mid+1; j1<=mid && j2
<= high; ) {
   if( aux[j1] <= aux[j2] )
      array[i++] = aux[j1++];
   else
      array[i++] = aux[j2++];
}
while( j1 <= mid )
   array[i++] = aux[j1++];
while( j2 <= high )
   array[i++] = aux[j2++];

//Sort:
int mid = low + (high-low)/2;
sort( array, low, mid, aux );
sort( array, mid+1, high, aux );
merge(array, low, mid, high, aux);
```

Tim Sort: (NI/S)
Merge sort with binary insertion sort
Best: ~n

Quick Sort: (I/NS)
Time: ~n log n Space: O(1)
Best: ~2n ln n Worst: $\sim n^2/2$

```
//Partition:
int i = l, j = r + 1;
while(true){
 while(a[++i] <= a[l])
   if(i == r)
     break;
 while(a[l] <= a[--j])
   if(j == l)
     break;
   if(i >= j)
     break;
   exchange(a,i,j);
 }
exchange(a,l,j);
return j;

//Sort:
if(l >= r)
 return;
int j = partition(arr,l,r);
sort(arr,l,j - 1);
sort(arr,j + 1,r);
```

Multi-pivot quick sort performs better because fewer cache misses.

3-Way Quick Sort: (I/NS)
Improve quick sort when duplicated keys.
Best: ~n

```
int mid = arr[(r + l) >> 1],i = l,j = r;
do{
   while(arr[i] < mid)
      i++;
   while(arr[j] > mid)
      j--;
   if(i <= j) {
      exchange(arr, i, j);
      i++;
      j--;
   }
}while(i <= j);
if(l < j)
   quickSort(arr,l,j);
if(r > i)
   quickSort(arr,i,r);
```

Priority Queue:
Binary heap:
Insert: $\log_2 n$ Pop: $\log_2 n$
P-ary heap:
Insert: $\log_p n$ Pop: $p \log_p n$

```
//Binary Max PQ
//Initialize
pq = (Key[]) new Comparable[Capacity + 1];
//isEmpty()
return n == 0;
//insert(Key x)
pq[++n] = x;
swim(n);
//swim(int k)
while(k > 1 && less(k / 2,k)){
 exch(k,k / 2)
 k = k / 2;
}
//sink(int k)
while(2 * k <= n){
 int j = 2 * k;
 if(j < n && less(j,j + 1))
   j++;
 if(!less(k,j)
   break;
 exch(k,j);
 k = j;
}
//pop() //delMax()
Key max = pq[1];
exch(1,n--);
sink(1);
```

Heap Sort: (I/NS)
Time: 2n log n Best: ~3n

```
//sort
int n = a.length;
MaxPQ<String> pq = new MaxPQ<String>();
for (int i = 0; i < n; i++)
    pq.insert(a[i]);
for (int i = n-1; i >= 0; i--)
    a[i] = pq.delMax();
//in-place sort
int n = a.length;
//Establishing heap O(n log n)
for (int k = n/2; k >= 1; k--)
    sink( a, k, n);
//Sort down O(n log n)
while (n > 1) {
   exch( a, 1, n);
   sink( a, 1, --n);
}
```

Intro Sort: (I/NS)
Quick sort →
(Stack depth exceeds 2lgn) Heap sort → (n = 16) Insertion sort

Symbol Tables: Key-Value Pair Abstraction:
//**contains**(Key key)
return get(key) != null;
//**delete**(Key key)
put(key,null);

Linked List Implementation:
search: ~n/2 insert: ~n min/max: ~n
floor/ceiling/rank: ~n select: ~n
ordered iteration: ~n log n
delete: ~n/2
key interface: equals()

Binary Search on Ordered Array:
search: ~log n insert: ~n/2 min/max: ~1
floor/ceiling/rank: ~log n select: ~1
ordered iteration: ~n delete: ~n/2
key interface: compareTo()

Binary Search Tree:
search: ~1.39lg n, worst ~n (h)
insert: ~1.39lg n, worst ~n (h)
min/max/floor/ceiling/rank/select: ~h
ordered iteration: ~n delete: sqrt(n)
//**get** (Key key)
Node x = root;
while(x != null){
  int cmp = key.compareTo(x.key);
  if(cmp < 0) x = x.left;
  else if(cmp > 0) x = x.right;
  else return x.val;
}
return null;
//**put** (Key key, Val val)
root = put(root,key,val);
//**put** (Node x, Key key, Val val)
if(x = null) return new Node(key,val);
int cmp = key.compareTo(x.key);
if(cmp < 0)
  x.left = put(x.left, key, val);
else if(cmp > 0)
  x.right = put(x.right, key, val);
else
  x.val = val;
return x;
//**floor** (Key key)
Node x = floor(root,key);
if(x == null) return null;
return x.key;
//**floor** (Node x, Key key)
if(x == null) return null;
int cmp = key.compareTo(x.key);
if(cmp == 0) return x;
if(cmp < 0) return floor(x.left,key);
Node t = floor(x.right,key);
if(t != null) return t;
return x;
//**rank** (Key key)(count all keys < k)
return rank(key, root);
//**rank** (Key key, Node x)
if(x == null) return 0;
int cmp = key.compareTo(x.key);
if(cmp < 0) return rank(key,x.left);
else if(cmp > 0) return 1 + size(x.left) +
rank(key,x.right);
else return size(x.left);
//**inorder** (Node x, Queue<Key> q)
if(x == null) return;
inorder(x.left,q);
q.entry(x.key);
inorder(x.right,q);
//**delete** (Key key)
root = delete(root,key);
//**delete** (Node x, Key key)
if(x == null) return null;
int cmp = key.compareTo(x.key);
if(cmp < 0) x.left = delete(x.left, key);
else if(cmp > 0) x.right = delete(x,right, key);
else{
  if(x.right == null) return x.left;
  if(x.left == null) return x.right;
  Node t = x;
  x = min(t.right);
  x.right = deleteMin(t.right);
  x.left = t.left;
}
x.count = size(x.left) + size(x.right) + 1;
return x;

2-3 Tree:
search/insert/delete: ~c lg n

Left Leaning Red-Black BST:
search/insert/delete: ~1.0lg n
worst: ~2lg n
//**get:** same as regular BST
//**isRed** (Node x)
return (x == null) ? false : (x.color == RED);
//**rotateLeft** (Node h)
assert isRed(h).right;
Node x = h.right;
h.right = x.left;
x.left = h;
x.color = h.color;
h.color = RED;
return x;
//**rotateRight** (Node h)
assert isRed(h.right);
Node x = h.left;
h.left = x.right;
x.right = h;
x.color = h.color;
h.color = RED;
return x;
//**flipColors** (Node h)
assert !isRed(h);
assert isRed(h.left) && isRed(h.right);
h.color = RED;
h.left.color = BLACK;
h.right.color = BLACK;
//**put** (Node h, Key key, Value val) {
if (h == null) return new Node(key, val, RED);
int cmp = key.compareTo(h.key);
if (cmp < 0) h.left = put(h.left, key, val);
else if (cmp > 0) h.right = put(h.right, key, val);
else if (cmp == 0) h.val = val;
if (isRed(h.right) && !isRed(h.left))
  h = rotateLeft(h);
if (isRed(h.left) && isRed(h.left.left)) h =
  rotateRight(h);
if (isRed(h.left) && isRed(h.right))
  flipColors(h);
return h;

Directed Graph:

Edge Lists:
space: E insert edge: 1 edge search: E
adjacency matrix: space: E^2 search: 1
  insert: 1(no parallel edges)
adjacency list: space: E+V insert: 1
  search: outdegree(v)

Adjacency List Implementation:
private final int V;
private final Bag<Integer>[] adj;
//**Digraph** (int V)
this.V = V;
adj = (Bag<Integer>[]) new Bag[V];
for (int v = 0; v < V; v++)
  adj[v] = new Bag<Integer>();
//**addEdge** (int v, int w)
adj[v].add(w);
//**adj** (int v)
return adj[v];

Depth-First Search:
private boolean[] marked;
//**DirectedDFS** (Digraph G, int s)
marked = new boolean[G.V()];
dfs(G, s);
//**dfs** (Digraph G, int v)
marked[v] = true;
for (int w : G.adj(v))
  if (!marked[w]) dfs(G, w);
//**visited** (int v)
return marked[v];

Connected Component:
private boolean marked[];
private int[] id;
private int count;
//**CC**(Graph G)
marked = new boolean[G.V()];
id = new int[G.V()];
for (int v = 0; v < G.V(); v++) {
  if (!marked[v]) {
    dfs(G, v);
    count++;
  }
}
//**dfs**(Graph G, int v)
marked[v] = true;
id[v] = count;
for (int w : G.adj(v))
  if (!marked[w])
    dfs(G, w);
//**connected**(int v, int w)
return id[v] == id[w];

Strong Component:
private boolean marked[];
private int[] id;
private int count;
//**KosarajuSharirSCC** (Digraph G)
marked = new boolean[G.V()];
id = new int[G.V()];
DepthFirstOrder dfs = new
DepthFirstOrder(G.reverse());
for (int v : dfs.reversePostorder()) {
  if (!marked[v]) {
    dfs(G, v);
    count++;
  }
}
//**dfs** (Digraph G, int v)
marked[v] = true;
id[v] = count;
for (int w : G.adj(v))
  if (!marked[w])
    dfs(G, w);
//**stronglyConnected** (int v, int w)
return id[v] == id[w];

Build Weighted Edge:
private final int v, w;
private final double weight;
//**Edge** (int v, int w, double weight)
this.v = v;
this.w = w;
this.weight = weight;
//**either** ()
return v;
//**other** (int vertex)
if (vertex == v) return w;
else return v;
//**compareTo** (Edge that)
if (this.weight < that.weight) return -1;
else if (this.weight > that.weight) return +1;
else return 0;

Build Weighted Graph by Adjacency List:
private final int V;
private final Bag<Edge>[] adj;
//**EdgeWeightedGraph** (int V) {
this.V = V;
adj = (Bag<Edge>[]) new Bag[V];
for (int v = 0; v < V; v++)
  adj[v] = new Bag<Edge>();
//**addEdge** (Edge e)
int v = e.either(), w = e.other(v);
adj[v].add(e);
add edge to both
adjacency lists
adj[w].add(e);
//**adj** (int v)
return adj[v];

Minimum Spanning Tree (Kruskal):
Build Priority Queue: frequency 1 time E
Delete Min: frequency E time log E
Union: frequency V time log V
Connected: frequency E time log V
Worst Case: ~E log E

private Queue<Edge> mst
    = new Queue<Edge>();
//**KruskalMST** (EdgeWeightedGraph G)
MinPQ<Edge> pq
    = new MinPQ<Edge> (G.edges());
UF uf = new UF(G.V());
while (!pq.isEmpty() && mst.size() < G.V()-1) {
  Edge e = pq.delMin();
  int v = e.either(), w = e.other(v);
  if (!uf.connected(v, w)) {
    uf.union(v, w);
    mst.enqueue(e);
  }
}
//**edges** ()
return mst;

Minimum Spanning Tree (Prim):
Delete Min: frequency E time log E
Insert: frequency E time log E
Worst Case: E log E

private boolean[] marked;
private Queue<Edge> mst;
private MinPQ<Edge> pq;
//**LazyPrimMST** (WeightedGraph G) {
pq = new MinPQ<Edge>();
mst = new Queue<Edge>();
marked = new boolean[G.V()];
visit(G, 0);
while (!pq.isEmpty() && mst.size() < G.V() - 1) {
  Edge e = pq.delMin();
  int v = e.either(), w = e.other(v);
  if (marked[v] && marked[w]) continue;
  mst.enqueue(e);
  if (!marked[v]) visit(G, v);
  if (!marked[w]) visit(G, w);
}
//**visit** (WeightedGraph G, int v)
marked[v] = true;
for (Edge e : G.adj(v))
  if (!marked[e.other(v)])
    pq.insert(e);
//**mst** ()
return mst;

Least Significant Digit First String Sort(LSD):
Time: ~2W(N + R) Extra Space: N + R
//**sort** (String[] a, int W)
int R = 256;
int N = a.length;
String[] aux = new String[N];
for (int d = W-1; d >= 0; d--) {
key-indexed counting
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
count[a[i].charAt(d) + 1]++;
for (int r = 0; r < R; r++)
count[r+1] += count[r];
for (int i = 0; i < N; i++)
aux[count[a[i].charAt(d)]++] = a[i];
for (int i = 0; i < N; i++)
a[i] = aux[i];
}

e.g. To sort
{13,105,492,776,1,6,214,99,98,96,11,21}.
(Recognize 1→001, 13→013, etc.)

	Round 1	Round 2	Round 3
0		1 105 6	1 6 11 13 21 96 98 99
1	1 11 21	11 13 214	105
2	492	21	214
3	13		
4	214		492
5	105		
6	776 6 96		
7		776	776
8	98		
9	99	492 96 98 99	

Most Significant Digit First String Sort(MSD):
• Partition array into R pieces according to first character (use key-indexed counting).
• Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort).

Random: N $\log_R$ N Worst: 2W(N + R)
Space: N + DR (D: longest prefix match)
//**sort** (String[] a) {
aux = new String[a.length];
recycles aux[]array
but not count[]array
sort(a, aux, 0, a.length - 1, 0);
//**sort** (String[] a,String[] aux,int lo,int hi,int d)
if (hi <= lo) return;
key-indexed counting
int[] count = new int[R+2];
for (int i = lo; i <= hi; i++)
count[charAt(a[i], d) + 2]++;
for (int r = 0; r < R+1; r++)
count[r+1] += count[r];
for (int i = lo; i <= hi; i++)
aux[count[charAt(a[i], d) + 1]++] = a[i];
for (int i = lo; i <= hi; i++)
a[i] = aux[i - lo];
for (int r = 0; r < R; r++)
sort(a,aux,lo+count[r],lo + count[r+1] - 1,d+1);

3-Way String Quick Sort:
Random: 1.39N lg N Worst Case: 1.39WN lg R
Extra Space: log N + W
//**sort** (String[] a)
sort(a, 0, a.length - 1, 0);
//**sort** (String[] a, int lo, int hi, int d)
if (hi <= lo) return;
int lt = lo, gt = hi;
int v = charAt(a[lo], d);
int i = lo + 1;
while (i <= gt) {
int t = charAt(a[i], d);
if (t < v) exch(a, lt++, i++);
else if (t > v) exch(a, i, gt--);
else i++;
}
sort(a, lo, lt-1, d);
if (v >= 0) sort(a, lt, gt, d+1);
sort(a, gt+1, hi, d);

Brute Force Substring Search:
Worst Case: ~MN
int M = pat.length();
int N = txt.length();
for (int i = 0; i <= N - M; i++) {
int j;
for (j = 0; j < M; j++)
if (txt.charAt(i+j) != pat.charAt(j))
break;
if (j == M) return i;
}
return N; //Not Found

Knuth Morris Pratt Substring Search:
Deterministic Finite State Automaton(DFA):
//**KMP** (String pat) {
this.pat = pat;
M = pat.length();
dfa = new int[R][M];
dfa[pat.charAt(0)][0] = 1;
for (int X = 0, j = 1; j < M; j++) {
for (int c = 0; c < R; c++)
dfa[c][j] = dfa[c][X];
//copy mismatch cases
dfa[pat.charAt(j)][j] = j+1;
//set match case
X = dfa[pat.charAt(j)][X];
//update restart state
}
//**search** (String txt)
int i, j, N = txt.length();
for (i = 0, j = 0; i < N && j < M; i++)
j = dfa[txt.charAt(i)][j];
if (j == M) return i - M;
else return N;

	0	1	2	3	4	5
charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6

Boyer-Moore Algorithm:
//**buildRight** ()
right = new int[R];
for (int c = 0; c < R; c++)
right[c] = -1;
for (int j = 0; j < M; j++)
right[pat.charAt(j)] = j;
//**search** (String txt)
int N = txt.length();
int M = pat.length();
int skip;
for (int i = 0; i <= N-M; i += skip) {
skip = 0;
for (int j = M-1; j >= 0; j--) {
if (pat.charAt(j) != txt.charAt(i+j)) {
skip = Math.max(1,j - right[txt.charAt(i+j)]);
break;
}
}
if (skip == 0) return i;
return N;

Rabin-Karp Algorithm:
private long patHash; // pattern hash value
private int M; //Pattern length
private long Q; //Modulus
private int R; //Radix
private long RM1; // R^(M-1) % Q
//**RabinKarp** (String pat)
M = pat.length();
R = 256;
Q = longRandomPrime();
RM1 = 1;
for (int i = 1; i <= M-1; i++)
RM1 = (R * RM1) % Q;
patHash = hash(pat, M);
//**hash** (String key, int M)
long h = 0;
for(int j = 0;j < M;j++)
h = (h * R + key.charAt(j)) % Q;
return h;
//**search** (String txt)
int N = txt.length();
int txtHash = hash(txt, M);
if (patHash == txtHash) return 0;
for (int i = M; i < N; i++) {
txtHash =
(txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
txtHash = (txtHash*R + txt.charAt(i)) % Q;
if (patHash == txtHash) return i - M + 1;
}
return N;

Hashing String:
private int hash = 0;
cache of hash code
private final char[] s;
//**hashCode**()
int h = hash;
if (h != 0) return h;
for (int i = 0; i < length(); i++)
h = s[i] + (31 * h);
hash = h;
return h;

Hash Function:
//**hash** (Key key)
return (key.hashCode() & 0x7fffffff) % M;
Separate Chaining Symbol Table:
Search/Insert/Delete: ~3~5 Worst Case: ~N
private int M = 97; // number of chains
private Node[] st = new Node[M];
private static class Node {
private Object key;
private Object val;
private Node next;
...
}
//**get** (Key key)
int i = hash(key);
for (Node x = st[i]; x != null; x = x.next)
if (key.equals(x.key))
return (Value) x.val;
return null;
//**put** (Key key, Value val)
int i = hash(key);
for (Node x = st[i]; x != null; x = x.next)
if (key.equals(x.key)) { x.val = val; return; }
st[i] = new Node(key, val, st[i]);

Linear Probing Symbol Table:
Search/Insert/Delete: ~3~5 Worst Case: ~N
private int M = 30001;
private Value[] vals = (Value[]) new Object[M];
private Key[] keys = (Key[])
new Object[M];
//**get** (Key key)
for (int i = hash(key);keys[i]!=null;i=(i+1) % M)
if (key.equals(keys[i]))
return vals[i];
return null;
//**put** (Key key, Value val)
int i;
for (i = hash(key); keys[i] != null; i = (i+1) % M)
if (keys[i].equals(key))
break;
keys[i] = key;
vals[i] = val;

An algorithm is a sequence of unambiguous **Instructions** for solving a problem, i.e., for obtaining a required **Output** for any legitimate **Input** in a **Finite** amount of time.
A collection is a **Data Type** that stores a group of items. The main operation on a collection are **Insert** and **Delete** items. Which item to delete determines the nature of a collection. In a **Stack**, removing the item (**Pop**) is done in a **LIFO** way. In a queue, removing the item (**Dequeue**) is in a **FIFO** way. In a **Priority Queue**, removing the item(**DeleteMax/Min**) is done with the **Largest/Smallest** item.
Stacks and Queues are normally implemented by **Linked Lists/Resizing Arrays**, while Priority Queues are normally implemented by **Heaps**.
A sorting algorithm is **In-place** if it uses $\leq c \log n$ extra memory.

If $T_1(n) = O(f(n))$, $T_2(n) = O(f(n))$, then $T_1(n)$ is **not** necessarily equals $O(T_2(n))$.
Building a heap from an array of n items requires O(n log n) time.
For large input size, merge sort will **not** always run faster than insertion sort.(Same input)
To prevent too many recursive call for tiny sized array slice in merge sort or quick sort, in practice to enhance efficiency normally use cutoff to insertion sort when the length of slice is small enough.

Data Structures are **Objects** created to organize data used in computation, a way to store and organize data in order to facilitate **access** and **modifications**.

A **list** is a collection that remembers the order of its elements. A **set** is an unordered collection of unique elements.

In a directed graph, **in-degree** of a vertex is the number of edges directed to the vertex and **out-degree** of a vertex is the number of edges started from the vertex.The **degree** of a vertex is the number of edges connecting it.

Mathematical Model for Running Time:
$T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$
A = array access
B = integer add frequencies
C = integer compare
D = increment (depend on algorithm, input)
E = variable assignment

Key of Divide and Conquer:
Divide into smaller parts(sub problems)
Solve the smaller parts **recursively**
Merge the result of the smaller parts

Data structures
Objects created to organize data used in computation
A way to store and organize data in order to facilitate access and modifications

Scientific Method
Observe nature of the world.
Hypothesize a model consistent with the observation.
Predict event using the hypothesis.
Verify the prediction by making further observations.
Validate by repeating until the hypothesis and observations agree.

Experiments must be reproducible.
Hypotheses must be falsifiable.

System Independent Effects: Algorithm/Input
Dependent Effects: Hardware/Software/System

Typical Memory Usage

Type	Bytes	Type	Bytes
boolean	1	char[]	2n+24
byte	1	int[]	4n+24
char	2	double[]	8n+24
int	4	Object Reference	8
float	4	Padding	up to 8
long	8	Object	16+instance variables
double	8		

Shallow Memory Usage:
Don't count referenced objects.
Deep Memory Usage:
Count memory recursively for referenced object.

Shell Sort Example:
Input:
S H E L L S O R T E X A M P L E
13-sort:
P H E L L S O R T E X A M S L E
4-sort:
L E E A M H L E P S O L T S X R
1-sort:
A E E E H L L L M O P R S S T X

Merge Sort Example:
Input:
M E R G E S O R T E X A M P L E
Sort left half:
E E G M O R R S T E X A M P L E
Sort right half:
E E G M O R R S **A E E L M P T X**
Merge result:
A E E E E G L M M O P R R S T X

Quick Sort Example:
Input:
Q U I C K S O R T E X A M P L E
Shuffle:
K R A T E L E P U I M Q C X O S
Partition:
E C A I E **K** L P U T M Q R X O S
Sort left:
A C E E I K L P U T M Q R X O S
Sort Right:
A C E E I K **L M O P Q R S T U X**

Pre-Order: Father→Left Child→Right Child
In-Order: Left Child→Father→Right Child
Post-Order: Left Child→Right Child→Father