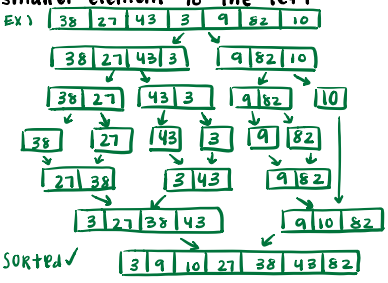
Merge Sort : comparison based.

Best : O(n); Avg/Worst : O(n log n)

This algorithm divides all data into as small possible sets then compares them, moving the smaller element to the left

Ex) 38 27 43 3 9 82 10



SORTED ✓

```
# Python program for implementation of MergeSort
def mergeSort(arr):
    if len(arr) >1:
        mid = len(arr)//2 #Finding the mid of the array
        L = arr[:mid] # Dividing the array elements
        R = arr[mid:] # into 2 halves

        mergeSort(L) # Sorting the first half
        mergeSort(R) # Sorting the second half

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i+=1
            else:
                arr[k] = R[j]
                j+=1
            k+=1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i+=1
            k+=1

        while j < len(R):
            arr[k] = R[j]
            j+=1
            k+=1
```

Counting Sort : specific range

O(n+k) ; n = # elements, k = range input

alg. based on keys between a specific range. Count # obj. with distinct key value. then calculate the position of each object in the output sequence.

Ex) array : 1 4 1 2 7 5 2

range: 0 - 8

index | 0 1 2 3 4 5 6 7 8

count | 0 2 2 0 1 1 0 1 0

then modify by adding i with i-1 and updating it

index | 0 1 2 3 4 5 6 7 8

value | 0 2 4 4 5 6 6 7

place | 1 2 3 4 5 6

then go through array and find placement by looking in index count for that value (aka 1 → 2) and then decrement index count place and since we end at 7, that means we will have 7 places.

place-array : 1 1 2 2 4 5 7

1 2 3 4 5 6 7

first element in array is 1, which has the place of 2 in the index count so place 1 at position 2 and decrement the place value. Repeat.

done.

```
# Python program for counting sort

# The main function that sort the given string arr[] in
# alphabetical order
def countSort(arr):

    # The output character array that will have sorted arr
    output = [0 for i in range(256)]

    # Create a count array to store count of inidividul
    # characters and initialize count array as 0
    count = [0 for i in range(256)]

    # For storing the resulting answer since the
    # string is immutable
    ans = ["" for _ in arr]

    # Store count of each character
    for i in arr:
        count[ord(i)] += 1

    # Change count[i] so that count[i] now contains actual
    # position of this character in output array
    for i in range(256):
        count[i] += count[i-1]

    # Build the output character array
    for i in range(len(arr)):
        output[count[ord(arr[i])]-1] = arr[i]
        count[ord(arr[i])] -= 1

    # Copy the output array to arr, so that arr now
    # contains sorted characters
    for i in range(len(arr)):
        ans[i] = output[i]
    return ans
```

Bubble Sort : comparison based.

Best = O(n); Avg./Worst = O(n²)

least efficient, iterates over every couplet, moving smaller element to the left and recursively doing this len-1 times.

Ex) 5 1 4 2 8 9 (length)

swap: 1 5 4 2 8 9 → 1 4 5 2 8 9 → 1 4 2 5 8 9

recursive: 1 4 2 5 8 9 (length-1)

recursive: 1 2 4 5 8 9

recursive: 1 2 4 5 8 9

recursive: 1 2 4 5 8 9 done! ✓

```
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Heap Sort: maxheap & remove

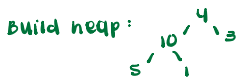
heapify = O(log n), create heap = O(n)

overall = O(n log n)

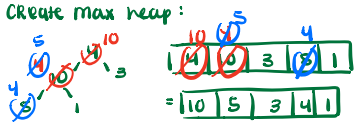
Have an array, then build a heap. Max-heapify it by comparing parent to child and swapping if child is larger. Then remove root and replace with last element and reflect this in the array and re-heapify. Repeat until only one element left in heap.

Ex) 4 10 3 5 1

Build heap :

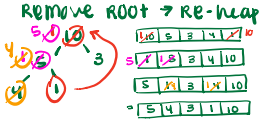


Create max heap :



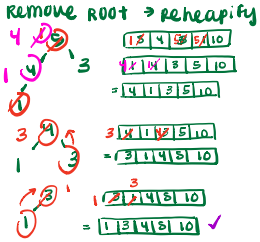
= 10 5 3 4 1

Remove root → re-heapify



= 5 4 3 1 10

Remove root → reheapify



= 1 3 4 5 10 ✓

only one element left so done sorting!

```
# Python program for implementation of heap Sort

# To heapify subtree rooted at index i.
# n is size of heap
def heapify(arr, n, i):
    largest = i # Initialize largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[i] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i] # swap

        # Heapify the root.
        heapify(arr, n, largest)

# The main function to sort an array of given size
def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)
```

lemma 8.3
Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in O(d(n+k)) time if the stable sort it uses takes O(n+k) time.

lemma 8.4
Given n b-bit numbers and any positive integer r<=b, RADIX-SORT correctly sorts these numbers in O((b/r)(n+2^r)) time if the stable sort it uses takes time O(n+k) time for inputs in the range 0 to k.

Quick Sort : divide & conquer.

Partition = O(n), Avg = O(n log n), Worst = O(n²)

"dual" version of merge sort → "un-merge" arr. break into "halves" of small elements (left) and large elements (right) then sort each side recursively and finally put both (sorted) sides together. can choose multiple "pivots", commonly last element. Even though Big O is avg/worse, this alg. is often faster in practice.

Ex) i starts at array[0]

j starts at array[i] and goes up to len-1

- if j is larger than pivot, cont. and j++
- if j is smaller than pivot, i++ and swap i and j

i | 10 16 8 12 15 6 3 9 5 | pivot

j → j → j → j → j → j → j → j

now 3 < 5 so increment i and swap (i, j)

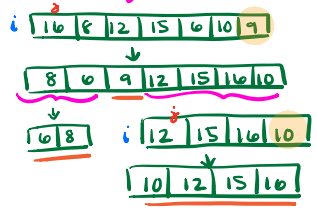
i | 3 16 8 12 15 6 10 9 5 | j → j = len-1 done.

when j = len-1, insert pivot at i+1 and repeat on both partitions (recursion).

3 5 16 8 12 15 6 10 9

less than pivot | greater than pivot

done



connected partitions :

3 5 6 8 9 10 12 15 16 ✓

```
# This function takes last element as pivot, places
# the pivot element at its correct position in sorted
# array, and places all smaller (smaller than pivot)
# to left of pivot and all greater elements to right
# of pivot
def partition(arr,low,high):
    i = ( low-1 )         # index of smaller element
    pivot = arr[high]     # pivot

    for j in range(low , high):

        # If current element is smaller than or
        # equal to pivot
        if   arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )

# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low  --> Starting index,
# high  --> Ending index

# Function to do Quick sort
def quickSort(arr,low,high):
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr,low,high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
```

Radix Sort: only sorts #'s

↺ (n log n)

sort based on the last digit (ones place) the 10's, 100's, etc. until done.

Ex) 170 45 75 90 802 24 2 66

170 90 80 2 2 24 45 75 66

2 802 24 45 66 170 75 90

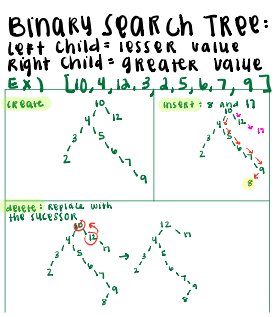
2 24 45 66 75 90 170 802 ✓

```
# Method to do Radix Sort
def radixSort(arr):

    # Find the maximum number to know number of digits
    max1 = max(arr)

    # Do counting sort for every digit. Note that instead
    # of passing digit number, exp is passed. exp is 10^i
    # where i is current digit number
    exp = 1
    while max1/exp > 0:
        countingSort(arr,exp)
        exp *= 10
```

```
    def insert(self, data):

        new_node = Node(data)  # This is the node we need to insert so we create an
        # instance of a node.

        if (self.__root == None): # If there is no root, then it becomes the root.
            self.__root = new_node
        else:
            current_node = self.__root # Since there is a root, we are going to start there,
            # and move down to find where we can insert the new_node.

            parent = None  # Keeping track of the parent of current node (so initially that is root
            # but this will change)

            while(current_node != None): # Wanting to find the location/position
            # where we can insert the new_node so we are looping through until the current position
            # is not null.
                current_node.setParent(current_node)
                parent = current_node

                if data < current_node.getData():
                    current_node = current_node.getLeftChild()
                else:
                    current_node = current_node.getRightChild()

            # We found the right position, and now we need to insert that new_node into
            # the correct position.
            if data < parent.getData():
                new_node.setParent(parent)
                current_node = parent
                current_node.setLeftChild(new_node)

            else:
                new_node.setParent(parent)
                current_node = parent
                current_node.setRightChild(new_node)
```

```
def delete(self, data):
    # Find the node to delete.
    # If the value specified by delete does not exist in the tree, then don't change the tree.
    # If you find the node and ...
    #  a) The node has no children, just set it's parent's pointer to Null.
    #  b) The node has one child, make the nodes parent point to its child.
    #  c) The node has two children, replace it with its successor, and remove
    #       successor from its previous location.
    # Recall: The successor of a node is the left-most node in the node's right subtree.
    # Hint: you may want to write a new method, findSuccessor() to find the successor when there are
      two children

    root = self.getRoot()

    deleted_node = self.deleteTheNode(root, data)


def deleteTheNode(self, root, data):
    # This is a function I created to assist the delete function so I can take in the current root
      (which
    # changes as you can see below) to accurently delete the desired node.

    if root == None:
        return root

    if data < root.getData():
        root.setLeftChild(self.deleteTheNode(root.getLeftChild(), data))

    elif data > root.getData():
        root.setRightChild(self.deleteTheNode(root.getRightChild(), data))
    else:
        if root.getLeftChild() == None: # If there is no left child, then that means there are either
            no children,
        # or it has just one right child.
            temp_node = root.getRightChild()
            root = None
            return temp_node
        elif root.getRightChild() == None: # No children.
            temp_node = root.getLeftChild()
            root = None
            return temp_node

        # This is a node with two children

        temp_node = self.findSuccessor(root.getRightChild())

        root.setData(temp_node.getData())

        # delete the successor node

        root.setRightChild(self.deleteTheNode(root.getRightChild(), temp_node.getData()))

    return root
```

```
def findSuccessor(self, aNode):
    # This is a new method to find the successor when there are two children and it does so
    # by looping through the node when it definitly has two children (aka when there is
    # a left child, you know there is an automatic right child so there are gaurenteed
    # two children) and then you keep going to the left child and checking.

    # Successor = smallest value greater than the node aka the next # after the node.

    current_node = aNode

    while(current_node.getLeftChild() != None):
        current_node = current_node.getLeftChild()
    return current_node
```




```
def _rb_Delete_Fixup(self, x):
    """
    This function receives a node, x, and fixes up the tree, balancing from x.

    Disclaimer: This code was based from the textbook.

    If x is red:
        change the color to black and done fixing.

    If x is black:
        transform the tree and move x up, until:
            x points to a red node or x is the root
    A key point in this function is to always set the color of x to black
    at the end of the function, which you can see on the last line.
    """
    while x is not self.root and x.color is "Black":
        if x is x.parent.leftChild:
            w = x.parent.rightChild
            if w.color is "Red":
                w.color = "Black"
                x.parent.color = "Red"
                self.leftRotate(x.parent)
                w = x.parent.rightChild
            if w.leftChild.color is "Black" and w.rightChild.color is "Black":
                w.color = "Red"
                x = x.parent
            else:
                if w.rightChild.color is "Black":
                    w.leftChild.color = "Black"
                    w.color = "Red"
                    self.rightRotate(w)
                    w = x.parent.rightChild
                w.color = x.parent.color
                x.parent.color = "Black"
                w.rightChild.color = "Black"
                self.leftRotate(x.parent)
                x = self.root

        else:
            w = x.parent.leftChild
            if w.color is "Red":
                w.color = "Black"
                x.parent.color = "Red"
                self.rightRotate(x.parent)
                w = x.parent.leftChild
            if w.rightChild.color is "Black" and w.leftChild.color is "Black":
                w.color = "Red"
                x = x.parent
            else:
                if w.leftChild.color is "Black":
                    w.rightChild.color = "Black"
                    w.color = "Red"
                    self.leftRotate(w)
                    w = x.parent.leftChild
                w.color = x.parent.color
                x.parent.color = "Black"
                w.leftChild.color = "Black"
                self.rightRotate(x.parent)
                x = self.root
    x.color = "Black"
```

```
def delete(self, key):
    # Same as binary tree delete, except we call rb_delete fixup at the end.
    # This is copied from the updated code from Matt on piazza.
    z = self.getNode(key)
    if z.leftChild is self.sentinel or z.rightChild is self.sentinel:
        y = z
    else:
        y = z.findSuccessor()

    if y.leftChild is not self.sentinel:
        x = y.leftChild
    else:
        x = y.rightChild

    if x is not self.sentinel:
        x.parent = y.parent

    if y.parent is self.sentinel:
        self.root = x

    else:
        if y == y.parent.leftChild:
            y.parent.leftChild = x
        else:
            y.parent.rightChild = x

    if y is not z:
        z.key = y.key

    if y.color == 'black':
        if x is self.sentinel:
            self._rb_Delete_Fixup(y)
        else:
            self._rb_Delete_Fixup(x)
        self._rb_Delete_Fixup(x)
```

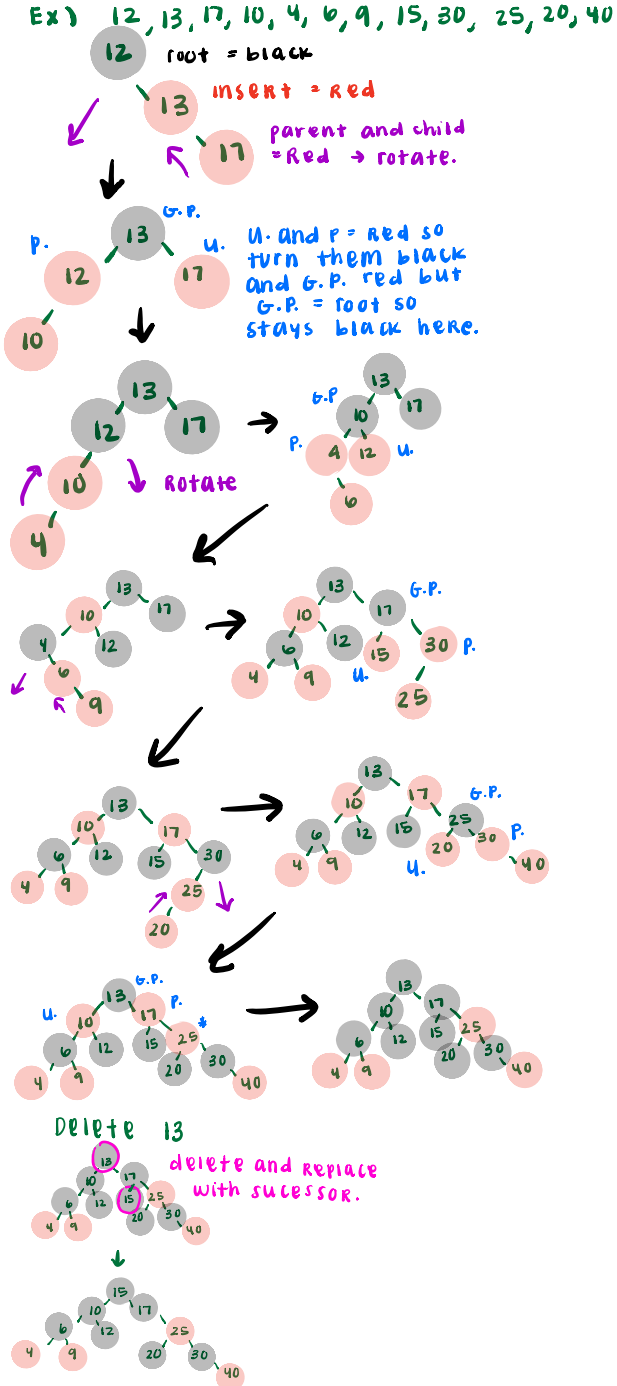
```
def leftRotate(self, currentNode):
    # perform a left rotation from a given node
    y = currentNode.rightChild
    currentNode.rightChild = y.leftChild

    if y.leftChild is not self.sentinel:
        y.leftChild.parent = currentNode

    y.parent = currentNode.parent

    if currentNode.parent is self.sentinel:
        self.root = y
    elif currentNode is currentNode.parent.leftChild:
        currentNode.parent.leftChild = y
    else:
        currentNode.parent.rightChild = y

    y.leftChild = currentNode
    currentNode.parent = y
```

```
def rightRotate(self, currentNode):
    """
    This function performs a right rotation from a given node
    """
    x = currentNode.leftChild
    currentNode.leftChild = x.rightChild

    if x.rightChild is not self.sentinel:
        x.rightChild.parent = currentNode

    x.parent = currentNode.parent

    if currentNode.parent is self.sentinel:
        self.root = x
    elif currentNode is currentNode.parent.rightChild:
        currentNode.parent.rightChild = x
    else:
        currentNode.parent.leftChild = x

    x.rightChild = currentNode
    currentNode.parent = x
```

Red Black Tree

Black height = log(n)
total height = 2 log (n)
Rules: • root is black
• insert Red
• same # black in each path
• if Red node has Red child, rotate
• if uncle & parent = Red, turn them black and the grandparent red.

Ex) 12, 13, 17, 10, 4, 6, 9, 15, 30, 25, 20, 40



Complexity perforance:

Big Ω ≤ Θ

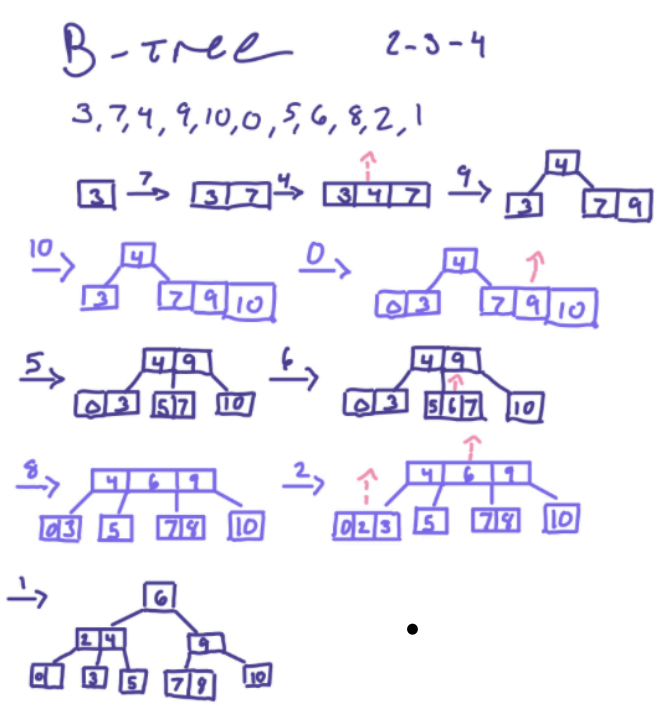
$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \dots < 2^n$

Big O ≥ Θ

lower bound — Avg. bound — upper bound

```python
def _rbInsertFixup(self, z):
    """
    Function to balance your tree after inserting

    Disclaimer: This code is based off of section 13.3 of the textbook.

    This is fixing the double red case after inserting as mentioned in the
    description of the insert function above.

    For the current node z, and it and it's parent is red and z
    is the unclude of y and we now have two cases to check:
        z is red ->
            color shift
            then check again for double red
        z is black ->
            rotate and then done
    """
    while z.parent.color == "Red":
        if z.parent == z.parent.parent.leftChild:
            y = z.parent.parent.rightChild
            if y.color == "Red":
                z.parent.color = "Black"
                y.color == "Black"
                z.parent.parent.color = "Red"
                z = z.parent.parent
            else:
                if z == z.parent.rightChild:
                    z = z.parent
                    self.leftRotate(z)
                z.parent.color = "Black"
                z.parent.parent.color = "Red"
                self.rightRotate(z.parent.parent)
        else:
            y = z.parent.parent.leftChild
            if y.color == "Red":
                z.parent.color = "Black"
                y.color = "Black"
                z.parent.parent.color = "Red"
                z = z.parent.parent
            else:
                if z == z.parent.leftChild:
                    z = z.parent
                    self.rightRotate(z)
                z.parent.color = "Black"
                z.parent.parent.color = "Red"
                self.leftRotate(z.parent.parent)

    self.root.color = "Black"
```

```python
def insert(self, key):
    """
    Add a key to the tree. Don't forget to fix up the tree and balance the nodes.

    Insert always needs to search to the bottom of the tree to find where where
    the new key (node z) should be.

    You put z there and color it read to maintain the black-height, but then
    you need to check if this causes a problem with two reds in a row, and
    if this is the case, then we fix this issue by color shifting/rotation which
    is seen later on.

    """
    z = RB_Node(key)  # creates node
    y = self.sentinel  # basically means null
    x = self.root

    if x is None:
        x = self.sentinel

    while x != self.sentinel:
        y = x
        if z.key < x.key:
            x = x.leftChild
        else:
            x = x.rightChild

    z.parent = y

    if y == self.sentinel:
        self.root = z

    elif z.key < y.key:
        y.leftChild = z

    else:
        y.rightChild = z

    z.leftChild = self.sentinel
    z.rightChild = self.sentinel
    z.color = "Red"
    self._rbInsertFixup(z)
```



	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

1. Suppose you have an array S of size n, where each element in S represents a different vote for class president, where each vote is given as an integer representing the student ID of the candidate. Without making any assumptions about who is running or how many candidates there are, design an $O(n \lg n)$ algorithm to determine which candidate receives the most votes. **[10 points]**
2. Given the input of the first problem, give an $O(n)$ time algorithm to determine if some candidate received a majority ($\lceil \frac{n+1}{2} \rceil$) of the votes.
   - $O(n)$ average or expected time is OK
   - so the hint is to look for the median

q1. Find the candidate who receives most number of votes.

Algo: Sort the array, and then count the number of similar ID students. After counting compare with maxium count and update it.

C++ code snippet:

```
int findMostVotedStudent(vector<int> &votes){
if (votes.length() == 0)
  return -1;

int maxVotes = 1;
int highestVotedStudentId = votes[0];

sort(votes.begin(), votes.end());

int count = 1;

for(int i=1; i<votes.size(); i++){
  if(votes[i] == votes[i-1]){
     count += 1;
  } else{
     if(maxVotes < count){
        maxVotes = count;
        highestVotedStudentId = votes[i-1];
     }
     count = 1;
  }
}
return highestVotedStudentId;
}
```

Time Complexity: O(n * logn)

Space Complexity: O(1)

There is a better solution to the above problem of Time Complexity O(n) and space O(n) using maps.

q2. Determine if is there any candidate with number of votes more than ceil($\frac{n+1}{2}$)

This is a classic problem known as majority element:

There is an O(n) solution to find out if there exists any number which occurs n/2+1 times.

The solution is simple, you need to keep the count of every students in a map and then see if any candidate has exceeded it.

c++ code snippet:

```
int determineMajorityElement(vector<int> &votes){
map<int, int> mp;
int total = votes.size();
int candidate = -1;

for(int i=0;i<n;i++){
  mp[votes[i]] += 1;

  if(mp[votes[i]] >= ceil(float(total+1)/2.0)){
     cout << "Found a candidate: "<<votes[i]<<" who has more than and equal to" <<ceil(float(total+1)/2.0)
<<" votes.\n";
     candidate = votes[i];
     return candidate;
  }
}
return candidate;
}
```

Timecomplexity: O(n)

Space complexity: O(n)

example

initialization: show that α is true after the <init> phase of the code has been executed

maintenance: show that if $\alpha \wedge \gamma$ is true, then α will be true after one execution of the loop body $\mathcal{L}$

termination: the loop finishes when γ is false, so argue that $\neg\gamma \wedge \alpha$ is the desired outcome

```
input: integer n>0
output: n(n+1)/2

--initialization
int s=0
int k=0

--loop
while k < n+1 do
    s = s+k
    k = k+1

--end
return s
```