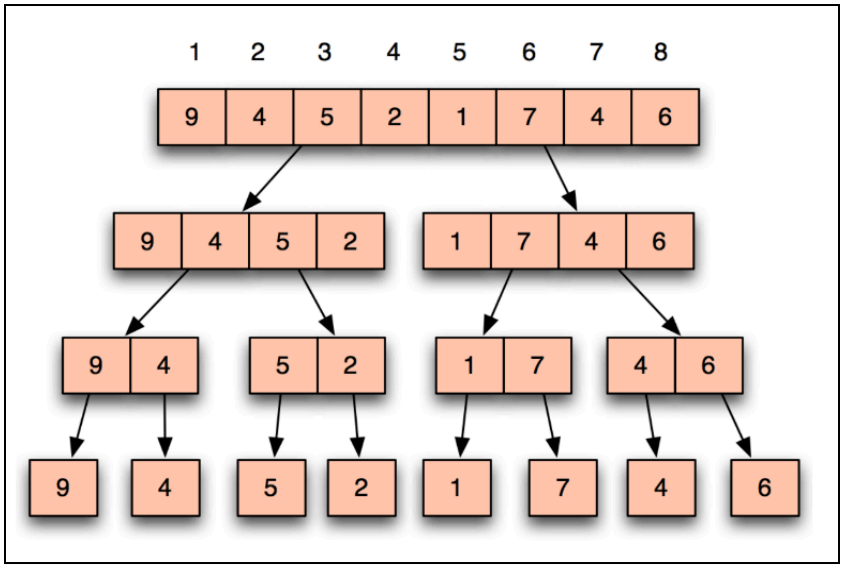
Sorting Algorithms

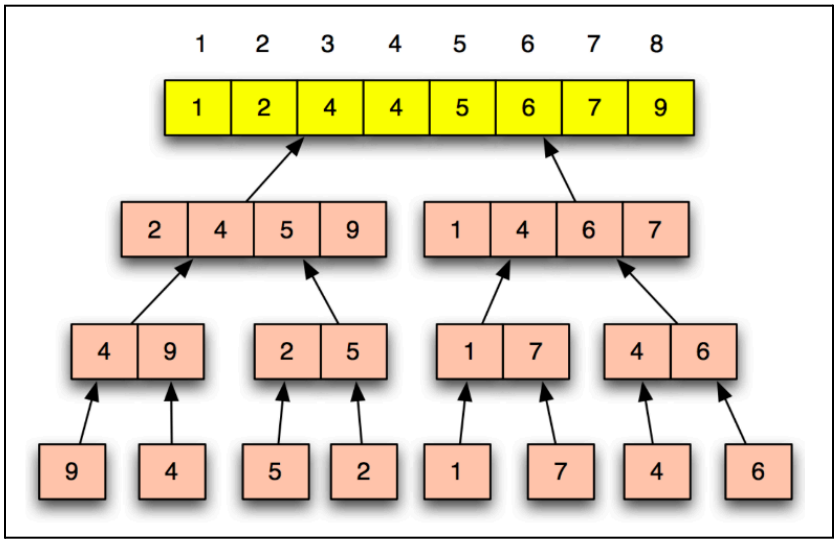
Merge Sort:

This sort follows a divide-and-conquer algorithm. It divides the problem into half, recursively sorts two subproblems, and then merges the results into a complete sorted sequence.

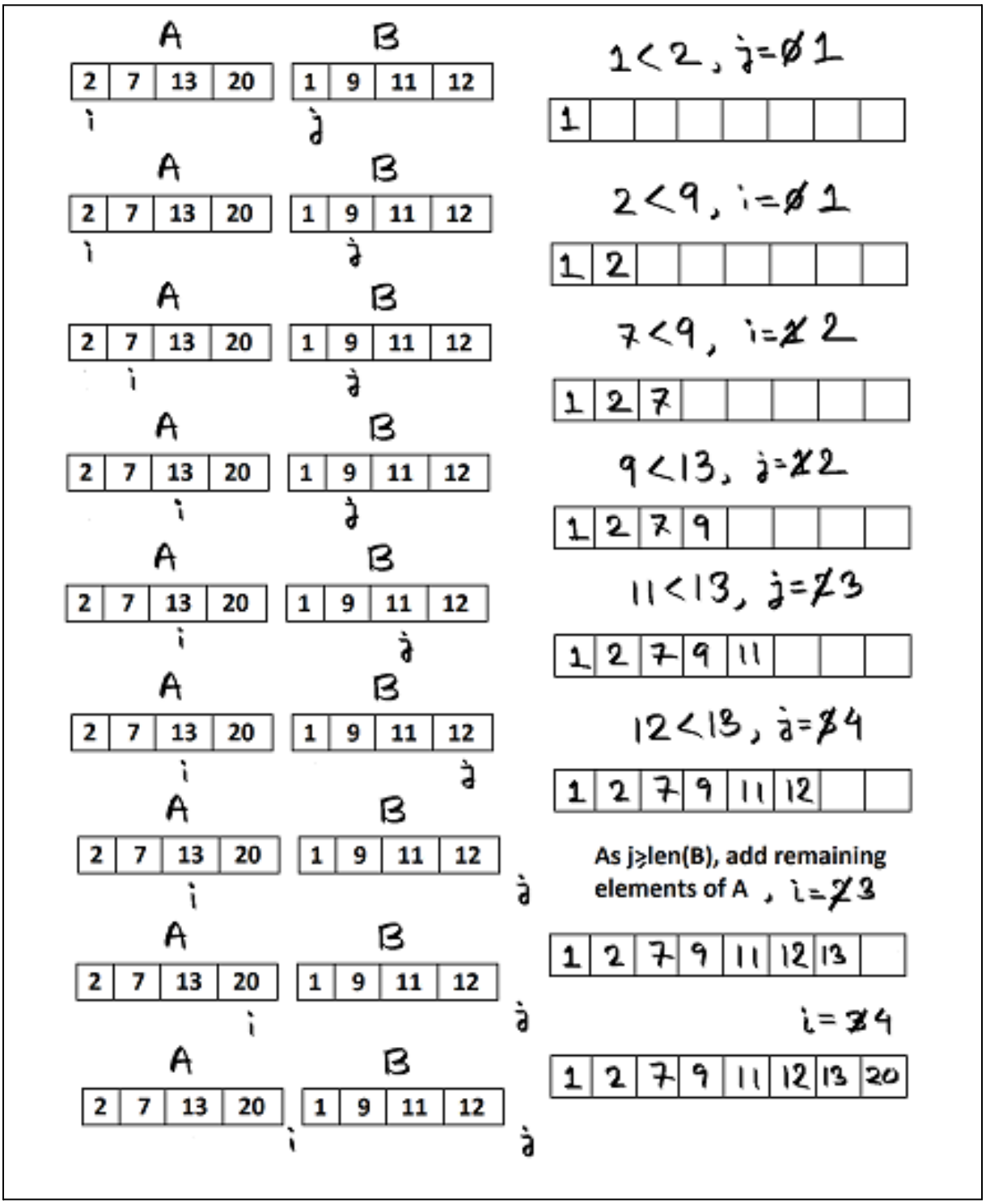
Divide:



Conquer & Combine:



Merging/Combining two sorted arrays A and B into another sorted array C in Θ(n):



Pseudo Code:

```
MERGE-SORT(A) ▷ A[0 : n]
1  if n = 1
2     then return
3     else                          ▷ recursively sort the two subarrays
4          A_1 = MERGE-SORT(A[0 : n//2])
5          A_2 = MERGE-SORT(A[ n//2 ] + 1 : n])
6          A = MERGE(A_1, A_2)       ▷ merge the sorted arrays

MERGE(A, B)
   INPUT: Two sorted arrays A and B
   OUTPUT: Returns C as the merged array
   ▷ n_1 = length[A], n_2 = length[B], n = n_1 + n_2
1  Create C of length n
2  Initialize two indices i and j to point to A and B respectively
3  while i < len(A) and j < len(B)
4        do Select the smaller of two and add to end of C
5           Advance the index that points to the smaller one
6  if i < len(A) or j < len(B)
7     then Copy the rest of the non-empty array to the end of C
8  return C
```

Advantages:

- Consistent time complexity O(n log n), making it efficient for large datasets
- Stable sorting algorithm, preserving the relative order of equal elements

Disadvantages:

- Requires additional memory space (Out-of-place sorting) for the temporary arrays during the merging process, leading to higher space complexity O(n)
- Slower for small datasets compared to simpler algorithms like insertion sort

Follow-up Question:

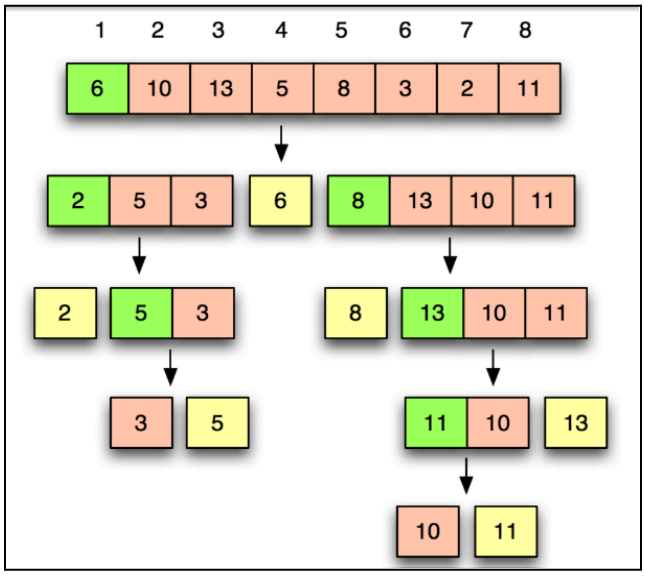
Q) What is the Recurrence Equation?

Ans: It depends on the number of subproblems, each subproblem size, and the work done at each step. In each step, there are 2 subproblems where size gets divided by 2 (n/2) and work done is n at each step. T(n) = 2T(n/2) + n

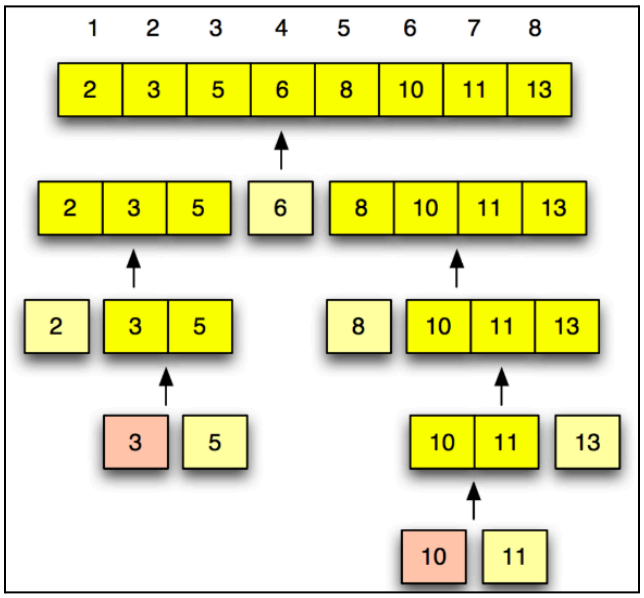
Quick Sort:

This sort also follows a divide-and-conquer algorithm. It partitions the array into subarrays around a pivot x such that the elements in the lower subarray ≤ x ≤ elements in the upper subarray, recursively sort the 2 subarrays, and then concatenates the lower subarray, pivot, and the upper subarray.

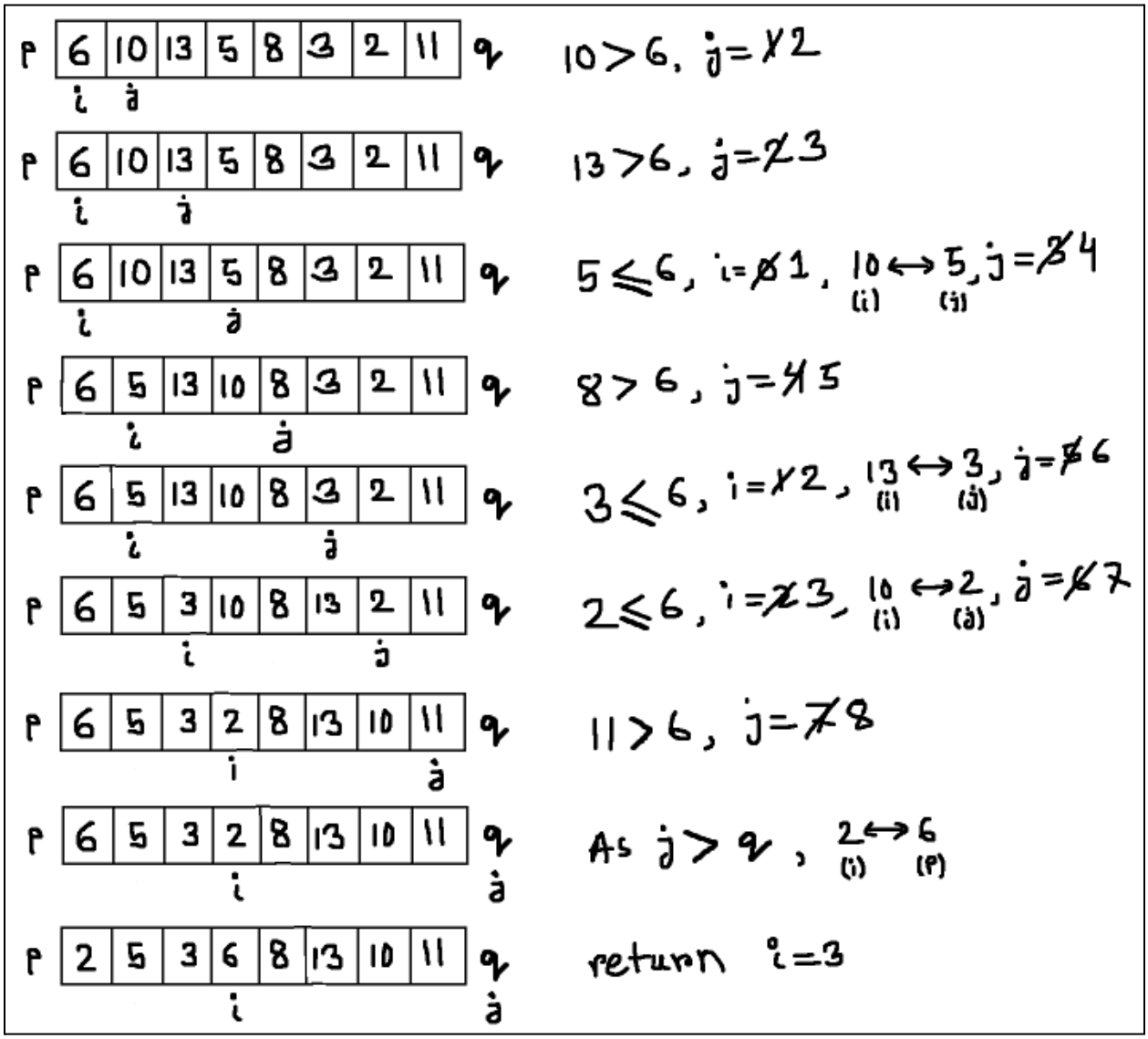
Divide / Partition:



Conquer & Combine:



Partitioning/Dividing an array A in Θ(n):



Pseudo Code:

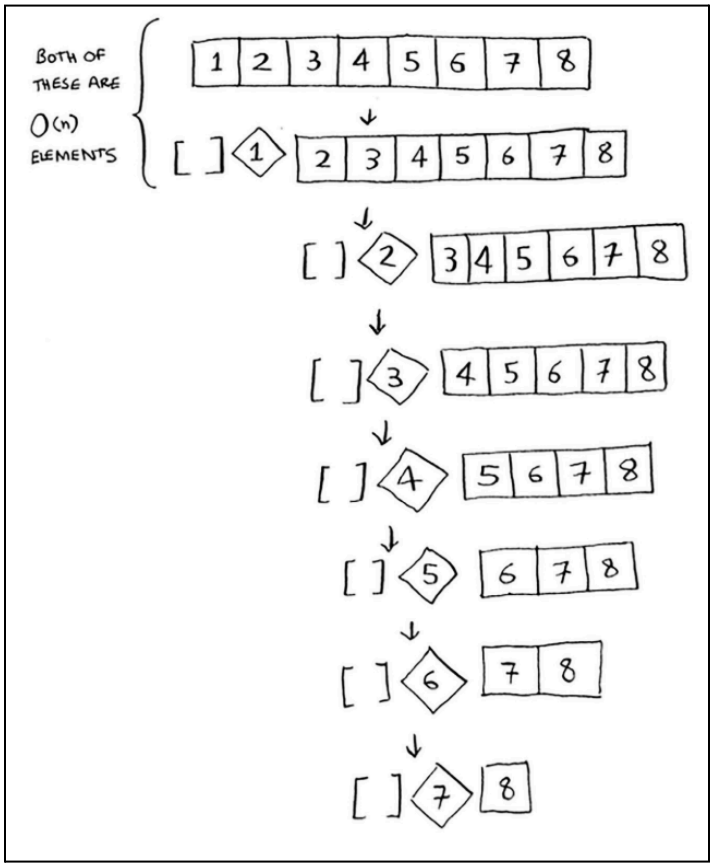
```
QUICKSORT(A, l, r) ▷ A[0 : n]
1  if l < r
2     then pivot ← PARTITION(A, l, r)
3          QUICKSORT(A, l, pivot − 1)
4          QUICKSORT(A, pivot + 1, r)

PARTITION(A, p, q) ▷ A[p : q]
1  x ← A[p]            ▷ pivot = A[p]
2  i ← p
3  for j ← p + 1 to q
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[p] ↔ A[i]
8  return i
```

Quicksort is unique because its speed depends on the pivot you choose.

Worst Case vs Average/Best Case:

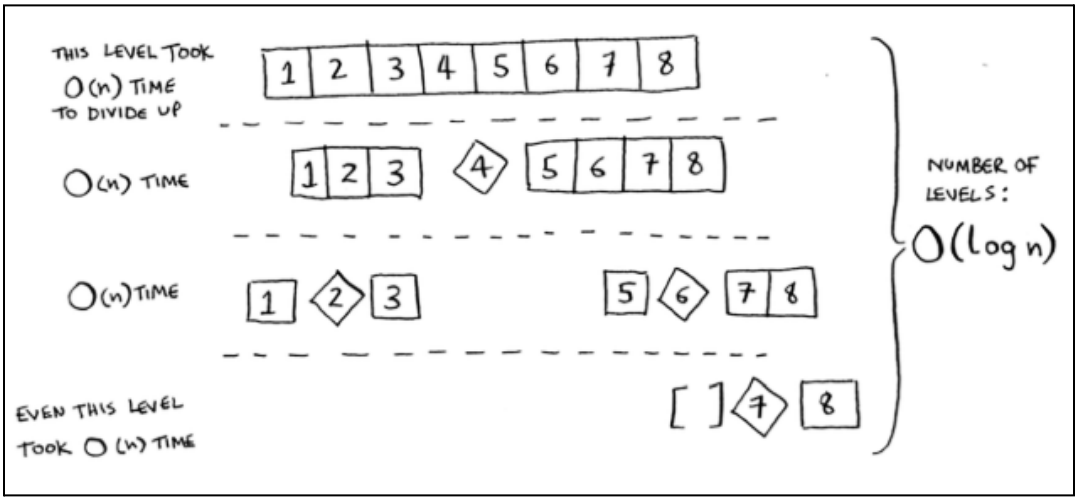
Worst Case happens when the pivot is the first or last element of a sorted (ascending or descending) array. The result is that one of the partitions is always empty.

Worst Case:



There are O(n) levels or the height is O(n), And each level takes O(n) time. The entire algorithm will take O(n) * O(n) = $O(n^2)$ time.

Best Case:



There are O(log n) levels or the height is O(log n). And each level takes O(n) time. The entire algorithm will take O(n) * O(log n) = O(n log n) time.

The best case is also the average case. If you always choose a random element in the array as the pivot, quicksort will complete in O(n log n) time on average

Advantages:

- Fastest sorting algorithm O(n log n), making it efficient for large datasets
- Does not require additional memory space (In-place sorting)

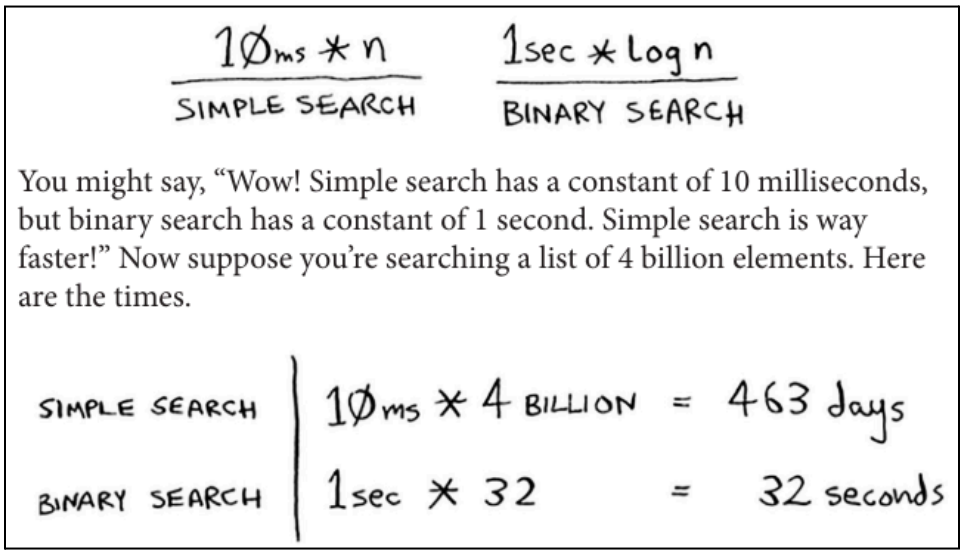
Disadvantages:

- $O(n^2)$ time complexity in worst-case scenario
- Unstable sorting algorithm, not preserving the relative order of equal elements

Follow-up Question:

Q) If quicksort is O(n log n) on average, but merge sort is O(n log n) always, why not use merge sort? Isn't it faster?

Ans: Even though both functions are the same speed in Big O notation, quicksort is faster in practice. When you write Big O notation like O(n), it means O(c * n) where c is some fixed amount of time that your algorithm takes. Let's see an example:



Quicksort has a smaller constant than merge sort. So if they're both O(n log n) time, quicksort is faster. And quicksort is faster in practice because it hits the average case way more often than the worst case.

Q) What is the Recurrence Equation when it is worst-case?

Ans: It depends on the number of subproblems, each subproblem size, and the work done at each step. In each step, there is 1 subproblem where size gets subtracted by 1 (n-1) and work done is n at each step. T(n) = T(n-1) + n

Q) What is the Recurrence Equation when it is best-case?

Ans: It depends on the number of subproblems, each subproblem size, and the work done at each step. In each step, there are 2 subproblems where size gets divided by 2 (n/2) and work done is n at each step. T(n) = 2T(n/2) + n

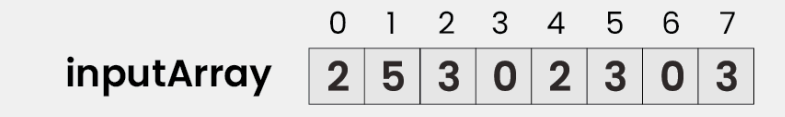
Q) How do you make sure your algorithm never reaches the worst-case when choosing 1st element as a pivot?

Ans: Use Randomized Quicksort. It is the same as the usual Quicksort but you will swap the pivot with a random number in the array within the range and then start partitioning. This will always give O(n log n).

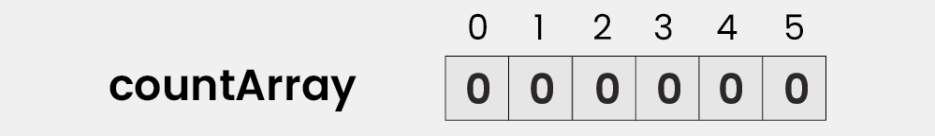
Count Sort:

Count sort, also known as counting sort, is a non-comparative integer sorting algorithm. This sorting technique does not perform sorting by comparing elements but by using a frequency array. It is efficient when the range of the input data, k (i.e., the difference between the maximum and minimum values) is not significantly greater than the number of elements to be sorted.

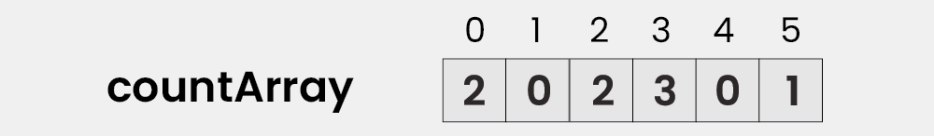
Step 1: Find out the range k (max - min +1) from the given array.



Step 2: Initialize a countArray[] of length k (5-0+1) with all elements as 0. This array will be used to store the occurrences of the elements of the input array.



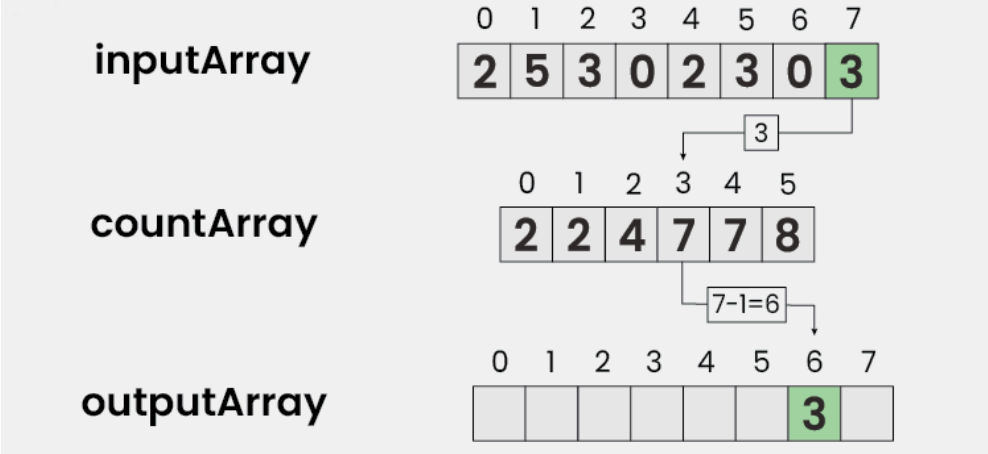
Step 3: In the countArray[], store the count of each unique element of the input array at their respective indices.



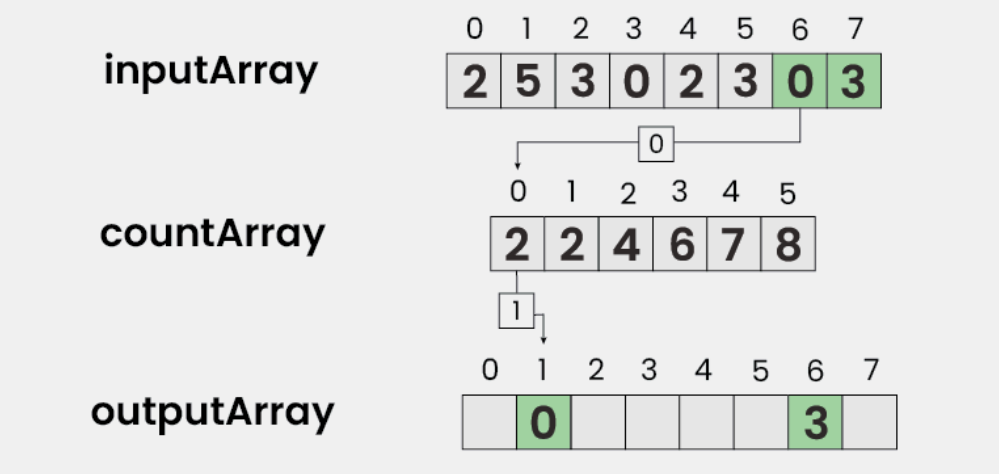
Step 4: Store the cumulative sum of the elements of the countArray[] by doing countArray[i] = countArray[i – 1] + countArray[i].

Step 5: Iterate from the end of the input array (to preserve stability) and update outputArray[countArray[inputArray[i]] – 1] = inputArray[i] and also, update countArray[inputArray[i]] = countArray[inputArray[i]] – 1 (so that duplicate values are not overwritten)

For i=7

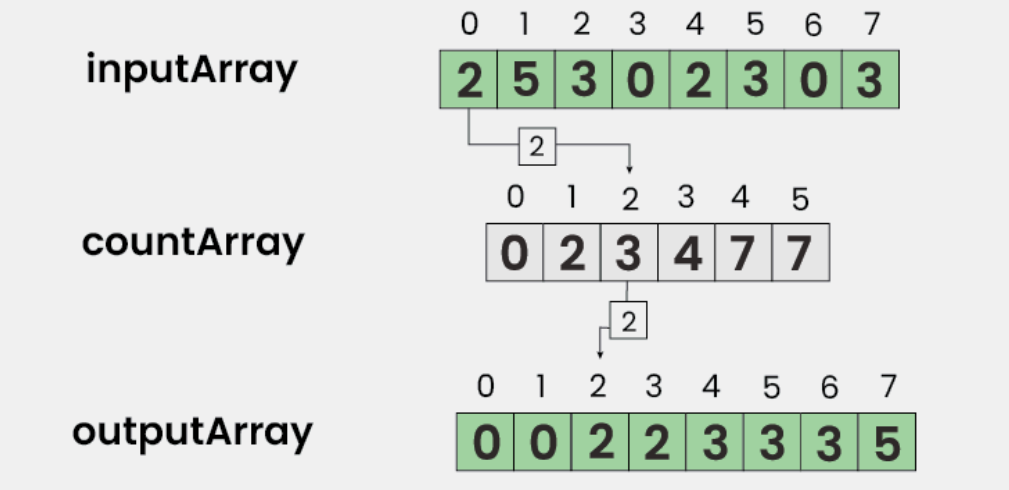


For i=6



.
.
.

For i=0



The outputArray will return a sorted array

Pseudo Code:

```
Counting-Sort(A,B,k)
    for i ←0 to k
        do C[i] ←0
    for j ←0 to length[A]-1
        do C[A[j]] ←C[A[j]]+1
    // C[i] contains number of elements equal to i.
    for i ←1 to k
        do C[i]=C[i]+C[i-1]
    // C[i] contains number of elements ≤ i.
    for j ←length[A]-1 downto 0
        do B[C[A[j]]-1] ←A[j]
           C[A[j]] ←C[A[j]]-1
```

Advantages:

- Consistent linear time complexity O(n+k)
- Stable sorting algorithm, preserving the relative order of equal elements

Disadvantages:

- Counting sort is inefficient if the range of values to be sorted is very large
- Requires additional memory space (Out-of-place sorting) for countArray and outputArray, leading to higher space complexity O(n+k)
- Counting sort does not work on decimal values

Follow-Up Question:

Q) What is the Time Complexity?

Ans: The first loop takes O(k), the second loop takes O(n), the third loop takes O(k) and the last loop takes O(n). Hence, time complexity is O(n+k)

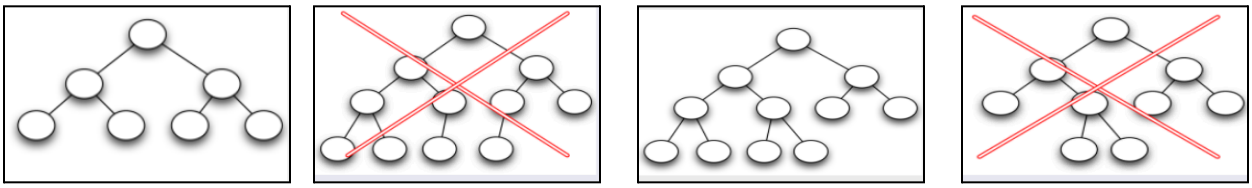
Heap Sort: [Optional]

Heap Sort uses a data structure called the heap.

Heap Data Structure:

Recap:

- In a Binary Tree, a parent can have a maximum of 2 nodes and a minimum of 0 nodes.
- In a Complete Binary Tree, nodes are added to the tree in a left-to-right manner not skipping any position. A parent can have 0, 1, or 2 children.



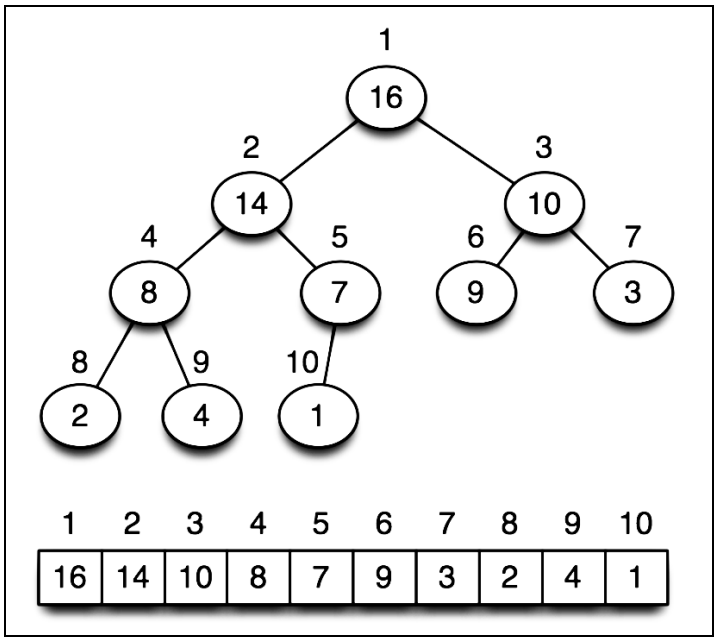
A Heap is an Abstract Data Type (ADT) for storing values. Its underlying data structure is an array.

A Heap has to be a complete binary tree and it must satisfy the heap property.
Heap property:

- The value of the parent must be greater than or equal to the values of the children (Max heap).
  or
- The value of the parent must be smaller than or equal to the values of the children. (Min heap).

There are two types of heaps. Max heap is mostly used (default). A heap can be either a max heap or a min heap but can't be both at the same time.

Heap data structure provides worst-case O(1) time access to the largest (max-heap) or smallest (min-heap) element and provides worst-case Θ(log n) time to extract the largest (max-heap) or smallest (min-heap) element.




```
MAXIMUM(A)
    return A[1]

PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

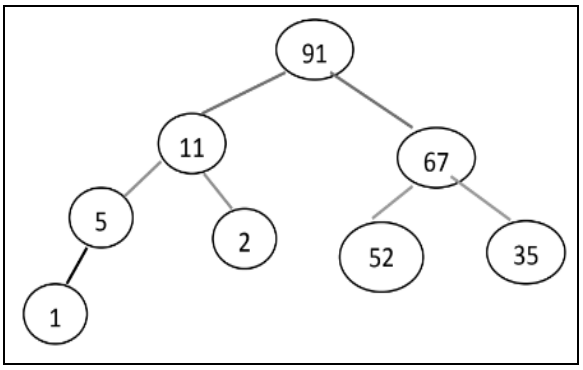
Note: Tree is used for efficient tracing. While programming, the data structure is a simple Array.

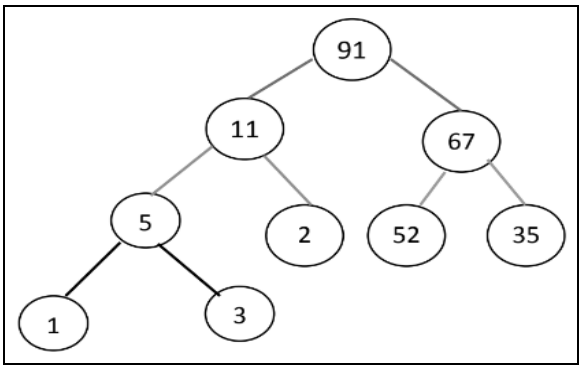
The benefit of using Array for Heap rather than Linked List is Arrays give you random access to its elements by indices. You can just pick any element from the Array by just calling the corresponding index. Finding a parent and their children is trivial. Whereas, Linked List is sequential. This means you need to keep visiting elements in the linked list unless you find the element you are looking for. Linked List does not allow random access as Array does. Also, each Linked List must have three (3) references to traverse the whole Tree (Parent, left, Right).

Heap Operations:

- Insert:

Inserts an element at the bottom of the Heap. Then we must make sure that the Heap property remains unchanged. When inserting an element in the Heap, we start from the left available position to the right.

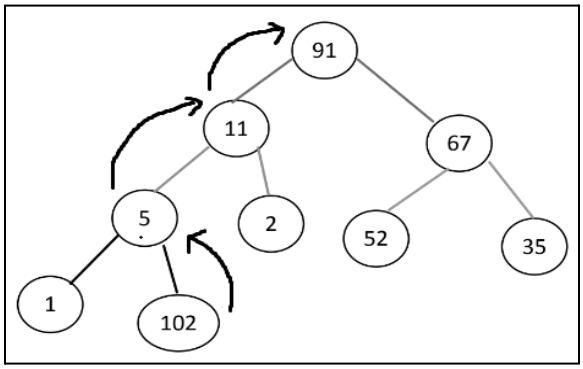


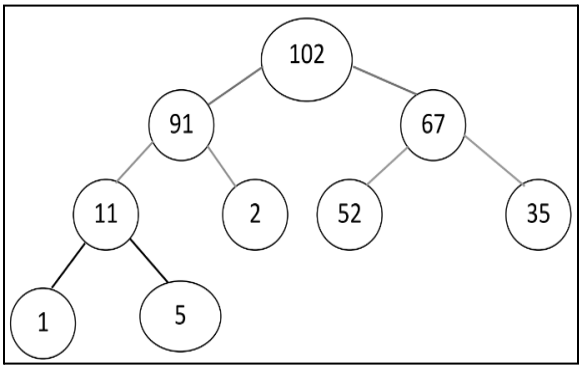


Here, Heap property is kept intact. What if we want to insert 102 instead of 3? 102 will be added as a child of 5 but the Heap property will be broken. Therefore, we need to put 102 in its correct position.

- HeapIncreaseKey / Swim:

Let the new node be ‘n’ (in this case it is the node that contains 102). Check ‘n’ with its parent. If the parent is smaller (n > parent) than the node ‘n’, replace ‘n’ with the parent. Continue this process until n is in its correct position.





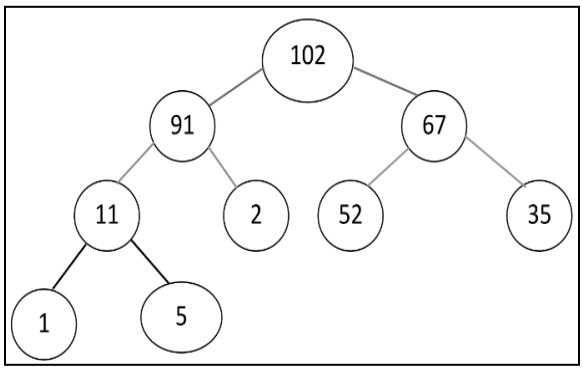
Best-case Time Complexity is O(1) when a key is inserted in the correct position at the first go.
Worst-case Time Complexity is when the newest node needs to climb up to the root
O(1) [insertion] + O(log n) [swim] = O(log n)

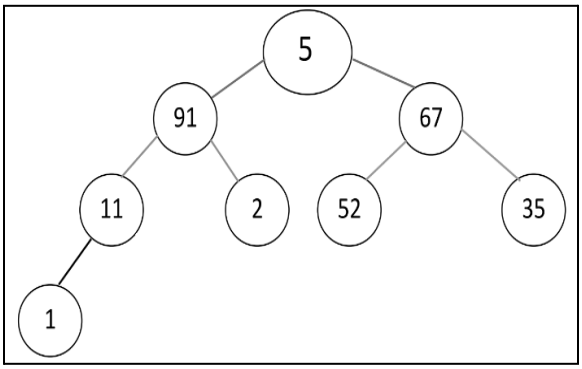
Pseudocode:

```
insert (H, key){
        size(H) = size(H) + 1;
        H[size] = key;
        swim (H, size);
}

swim(H, index){
        if (index < = 1){
                return;
        }else{
                parent = H[index/2];
                if (parent > H[index]){
                        return;
                }else{
                        exchange parent with H[index]
                        swim(H, parent);
                }
}
```

- Delete:

In heap, you cannot just cannot randomly delete an item. Deletion is done by replacing the root with the last element. The Heap property will be broken as small value will be at the top (root) of the Heap. Therefore we must put it in the right place.

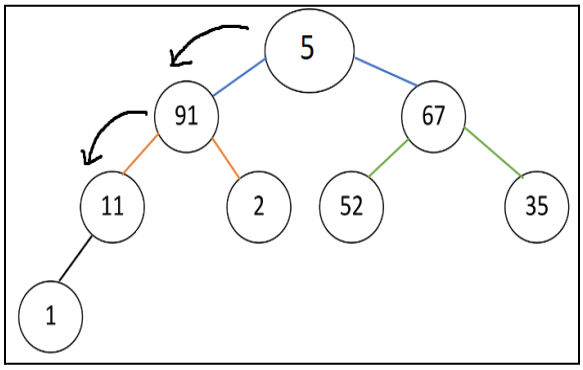


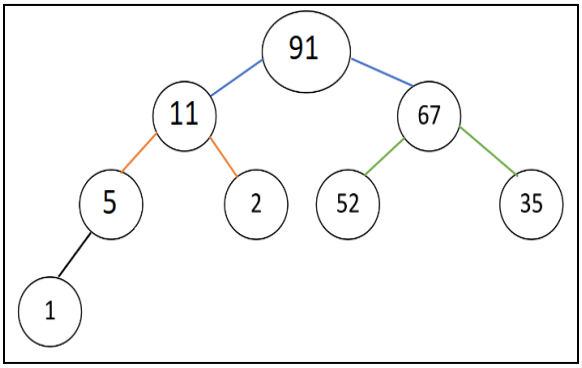


Here, the root 102 will be replaced by the last element 5, and 102 will be removed. Heap property will be broken. Therefore, we need to put 5 in its correct position.

- MaxHeapify / Sink:

Let the replaced node be ‘n’ (in this case it is the node that contains 5). Check ‘n’ with its children. If the node ‘n’ is smaller (n < any child) than any child, replace ‘n’ with the largest child. Continue this process until n is in its correct position.





The deleted element will always be the maximum element available in max-heap.Time Complexity is O(1) [deletion] + O(log n) [sink] = O(log n)

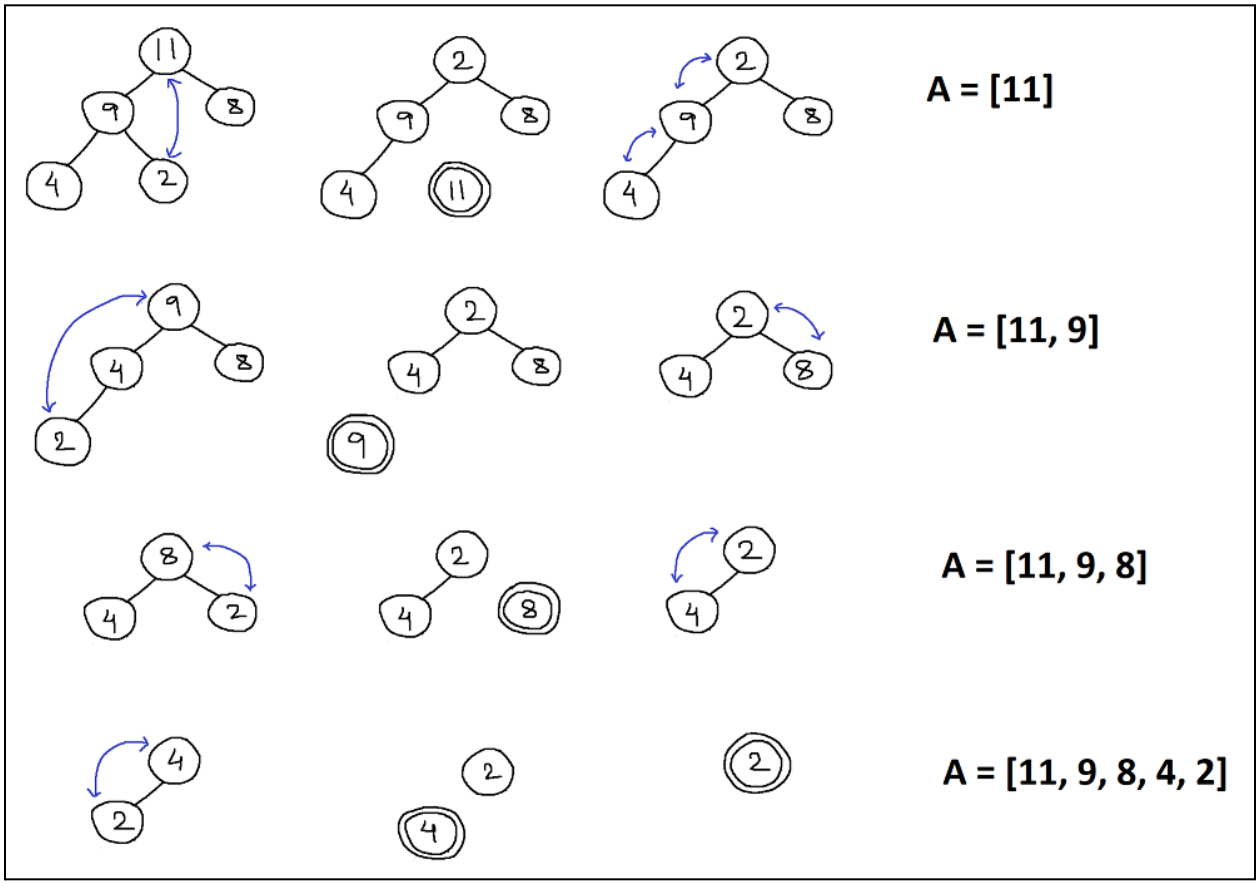
Pseudocode:

```
delete(H){
        if (size(H)==1){
                return;
        }else{
                exchange H[1] with H[size]
                size --;
                maxHeapify(H, 1)
            }
}

maxHeapify(H, index){
        if (size(H) ==1){
                return;
        }else{
                left = 2*index;
                right=2*index+1;
                if (left <= size && right<=size){
                        exchange H[1] with Max (H[left], H[right]);
                        maxHeapify(Max (left, right));
                }else{
                        if (left<= size && right>size){
                        exchange H[1] with (H[left]);
                    }
```

- Heap Sort:

Delete + Sink all the nodes of the heap and store them in an array. The array will return a sorted array in descending order. Reversing the array will give a sorted array in ascending order.

Simulation:



Delete + Sink takes O(log n) and for ‘n’ nodes, Heap Sort will take O(n log n).

Pseudocode:

```
for all nodes i = 1 to n{
        delete (H);
}
```

- Build Max Heap:

You are given an arbitrary array and you have been asked to build it a heap. This will take O(n log n).

Pseudocode:

```
for all nodes i = 1 to n{
        swim (H, i);
}
```

Advantages:

- Consistent time complexity O(n log n), making it efficient for large datasets
- Does not require additional memory space (In-place sorting)
- Often used as a priority queue

Disadvantages:

- Unstable sorting algorithm, not preserving the relative order of equal elements