**In my experience, from most to least commonly mentioned in interviews:  
1. quicksort: implement it, explain it  
2. mergesort: implement it, talk about space complexity as well as time complexity  
3. insertion sort: explain when it can be better than the above two  
4. heapsort: explain how it works, and how heaps work in general  
5. bubble sort: why it's awful  
6. radix/counting/bucket sort: when it's useful  
7. selection sort: usually thrown in as an example when asked to list sorting algorithms you know**

**Best Case -** Insertion Sort

**Average Case -** Quick Sort

**Worst Case -** Merge and Heap Sort but heap has better memory

* **Heap Sort -** Always
* **Insertion Sort -** When the array is almost sorted
* **Counting Sort -** Used when the largest term is of the order of n.
  + [5, 4, 7, 2, 100, 6, 8766567546758, 34] Counting sort will be terrible.
  + [20, 9, 8, 7, 6, 5, 4, 3] - Counting Sort will be amazing
  + If the largest term is of the order of n2 then use radix sort.
* **Radix Sort -** Use normally with integers in a case where the range of the numbers to be sorted is of the order n2. Integers are sorted from least significant digit to most significant digit.
* **Bucket Sort -** Bucket sort is used for sorting a distribution that is evenly distributed.
  + Example - 0 to 1 is divided into 10 buckets 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
  + Each number between 0 and 1 go into each of the buckets.

Java Sorting

* Simple Sorting: **Arrays.sort(arr);**
* Subarray Sorting: **Arrays.sort(arr, toIndex, fromIndex);**
  + toIndex is included in sorting but fromIndex is not
* Reverse Order sorting:
  + int has to be converted to Integer
  + **Integer[] arr1 = new Integer[arr.length];**
  + **for (int i = 0; i < arr.length; i++) arr1[i] = arr[i];**
  + **Arrays.sort(arr1, Collections.reverseOrder());**
  + **for (int i = 0; i < arr.length; i++) arr[i] = arr1[i];**
* Alphabet Simple Sorting: **Arrays.sort(arr);**
  + Alphabet Reverse Order sorting: **Arrays.sort(arr, Collections.reverseOrder());**
* **For sorting using comparator, Check the java file.**

Selection Sort

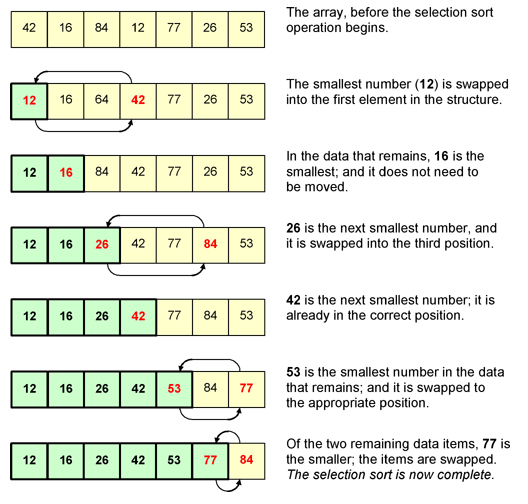
**How it works**

* Go from left to right, find the smallest number, and swap it with the first the first element.
* Keep doing this again and again until you swap the last element.
* After every pass, the list will start to get sorted from the start.
* You can ignore the starting element after every pass.
* First find the lowest element by going through 1 to n.
* Then find the second lowest element by going through 2 to n.
* **You swap in selection sort.**

**Analysis of Sorting Algorithm**

|  |  |  |  |
| --- | --- | --- | --- |
| Best Case Time | Worst Case Time | Average Case Time | Space Complexity |
| O(n2) | O(n2) | O(n2) | O(1) |

**Example of Sorting Algorithm**



**Notes:**

* This is one of the simplest and the most non-efficient sorting algorithm
* Never use

Bubble Sort

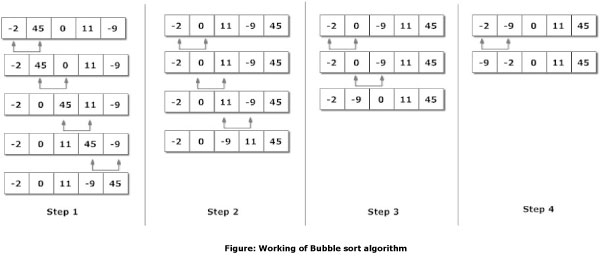
**How it works**

* Go from left to right and keep switching adjacent terms if the left one is bigger than right.
* If a number is exchanged then for the next comparison, the new places will be used and not the old places.
* Larger elements bubble up at the end of the series.
* The number of passes will be one less than the total no. of terms. E.g. In the figure below, for 5 terms, you get 4 passes.
* After each pass, the last term is fixed. You don’t need to consider the last term for the next pass. In the last pass, only the first 2 elements will be compared.
* After each pass, the series starts to get sorted from the end.

**Analysis of Sorting Algorithm**

|  |  |  |  |
| --- | --- | --- | --- |
| Best Case Time | Worst Case Time | Average Case Time | Space Complexity |
| O(n2) | O(n2) | O(n2) | O(1) |

**Example of Sorting Algorithm**



**Notes:**

* This is one of the simplest and the most non-efficient sorting algorithm

Insertion Sort

**How it works**

* Go from left to right and find the smallest element and take it out.
* Slide all the elements to right to make up place at the start of the series.
* Then you start searching from the next element as the first element is sorted.
* All the elements start to get sorted at the start of the series.
* **Here all the elements slide to make up space rather than swap.**

**Analysis of Sorting Algorithm**

|  |  |  |  |
| --- | --- | --- | --- |
| Best Case Time | Worst Case Time | Average Case Time | Space Complexity |
| O(n) | O(n2) | O(n2) | O(1) |

**Example of Sorting Algorithm**



**Notes:**

* If the series is sorted or close to sorted then this works the best.
* This is the fastest when array is sorted or almost sorted.

Quick Sort

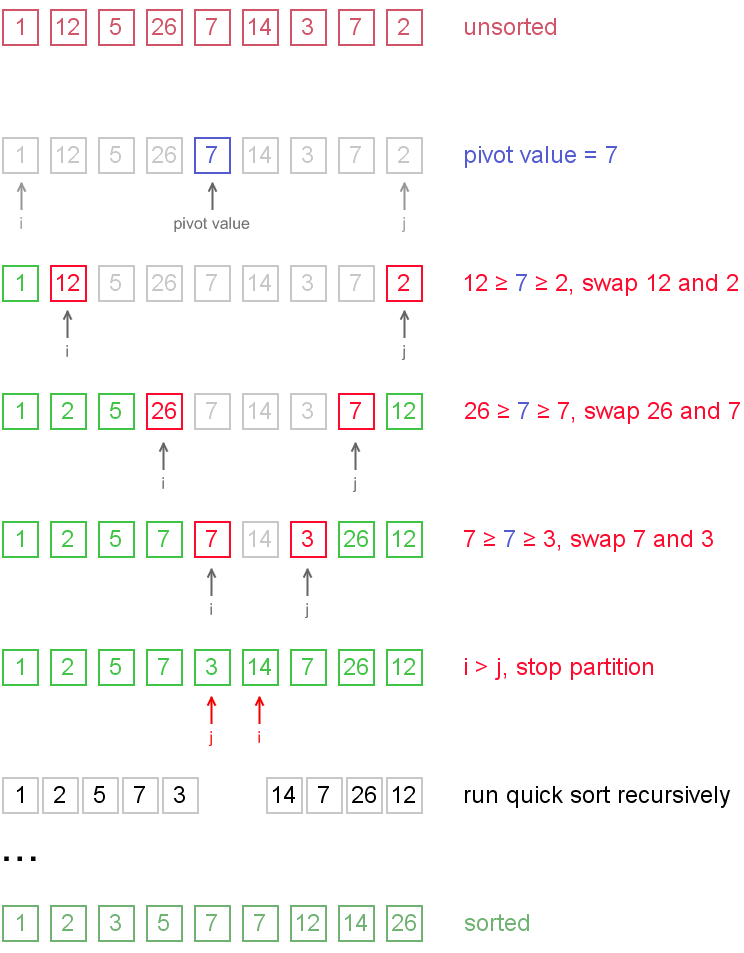
**How it works**

* Divide the array from the middle and maintain two indices I and J. I will be at first element of 1st array and J will be at the last of element of the second array.
* Now the middle point will be pivot.
* Now keep taking I to the right and J to the left. If the value at I index is greater than or equal to pivot then exchange it with the value at J index which is lower than or equal to pivot.
* This keeps going on until I and J cross.
* The 2 subarrays then divide further into 4 subarrays.
* The two subarrays are dividing where left array is from lo to wall and right array is wall to hi.
* The wall is decided as the point where the index1 crossed with index2 in the previous iteration of quickSort.
* This division continues until there is one element left in the subarray and there are many subarrays.
* The pivot is most often chosen in the middle.
* If the first and last element is chosen then there is a risk of worse case behavior.

**Analysis of Sorting Algorithm**

|  |  |  |  |
| --- | --- | --- | --- |
| Best Case Time | Worst Case Time | Average Case Time | Space Complexity |
| O(n log n) | O(n log n) | O(n2) | O(1) |

**Example of Sorting Algorithm**



**Notes:**

* Log n grows much slower than n and hence this is better.
* Worst case(O(n2)) is when the pivot is either the smallest or the biggest element.

**Code:**

**public** **class** Quick\_Sort {

**public** **static** **void** sort(**int**[] arr) {

*quick\_Sort*(arr, 0, arr.length - 1);

}

**public** **static** **void** quick\_Sort(**int**[] arr, **int** lo, **int** hi) {

**if** (hi <= lo + 1) **return**;

**int** pivot = arr[(hi + lo)/2];

**int** index1 = lo, index2 = hi;

**while** (index1 < index2) {

**if** ((arr[index1] >= pivot) && (arr[index2] <= pivot)){

*swap*(arr, index1, index2);

index1++;

index2--;

}

**else** {

**if** (arr[index1] < pivot) index1++;

**if** (arr[index2] > pivot) index2--;

}

}

// index1 is the wall

*quick\_Sort*(arr, lo, index1);

*quick\_Sort*(arr, index1 + 1, hi);

}

**public** **static** **void** swap(**int**[] arr, **int** index1, **int** index2) {

**int** temp = arr[index1];

arr[index1] = arr[index2];

arr[index2] = temp;

}

}

Iterative Quick Sort

**How it works**

* For iterative Quick Sort, a stack has to maintained.
* The stack has the left value and the right value of the array that has to be quick sorted.
* While the stack is not empty pop two values from the stack.
* The first one is the left index and the second one is the right index.
* Select the middle value and take it as pivot.
* Take the left index from left to right and right index from right to left and whenever you see a value on the left that is greater than pivot and you see a value on the right that is less than pivot then you swap them.
* You stop when they cross and send back the index at which the left index and right index cross each other. This index value is called the wall. This wall value is returned.
* Now if wall is greater than the left value then you push left value and wall - 1 onto the stack.
* If the wall is less than the right value then you push the wall + 1 and right value on the stack.
* Again continue doing the above steps until the stack is empty.

Merge Sort

**How it works**

* Merge sort uses an extra space of memory as big as the no. of elements. It is normally like 5n or 6n etc. but you take that as O(n).
* Keep dividing the array as long as there are multiple subarrays of 1 element.
* Now start combining all the arrays to make the bigger array. 2 one element array forms one 2 element array and so on.
* This is done by having 2 indices, I and J, at the start of the two subarrays. If I is smaller than J then I will go first. I will shift by one to the right. Now compare I and J again and whichever is smaller go again.
* This continues as the sizes of arrays become bigger and bigger.
* At the end, you get the sorted whole element.

**Analysis of Sorting Algorithm**

|  |  |  |  |
| --- | --- | --- | --- |
| Best Case Time | Worst Case Time | Average Case Time | Space Complexity |
| O(n log n) | O(n log n) | O(n log n) | O(n) |

**Example of Sorting Algorithm**

****

**Notes**

* This sorting algorithm uses the largest amount of memory and hence is not memory efficient.
* This sorting algorithm is the fastest even at worst conditions.
* Quicksort is comparable to merge ort in the average case but in worst case merge sort is much better

Bucket Sort

**How it works**

* Find the largest term and smallest term and divide it into a specific number of buckets.
* I divide it into 10 buckets only.
* Now in each bucket solve using insertion sort.
* Once you solve using insertion sort, combine all of them in a linear fashion. All the elements in first bucket, all the elements in second bucket and so on.

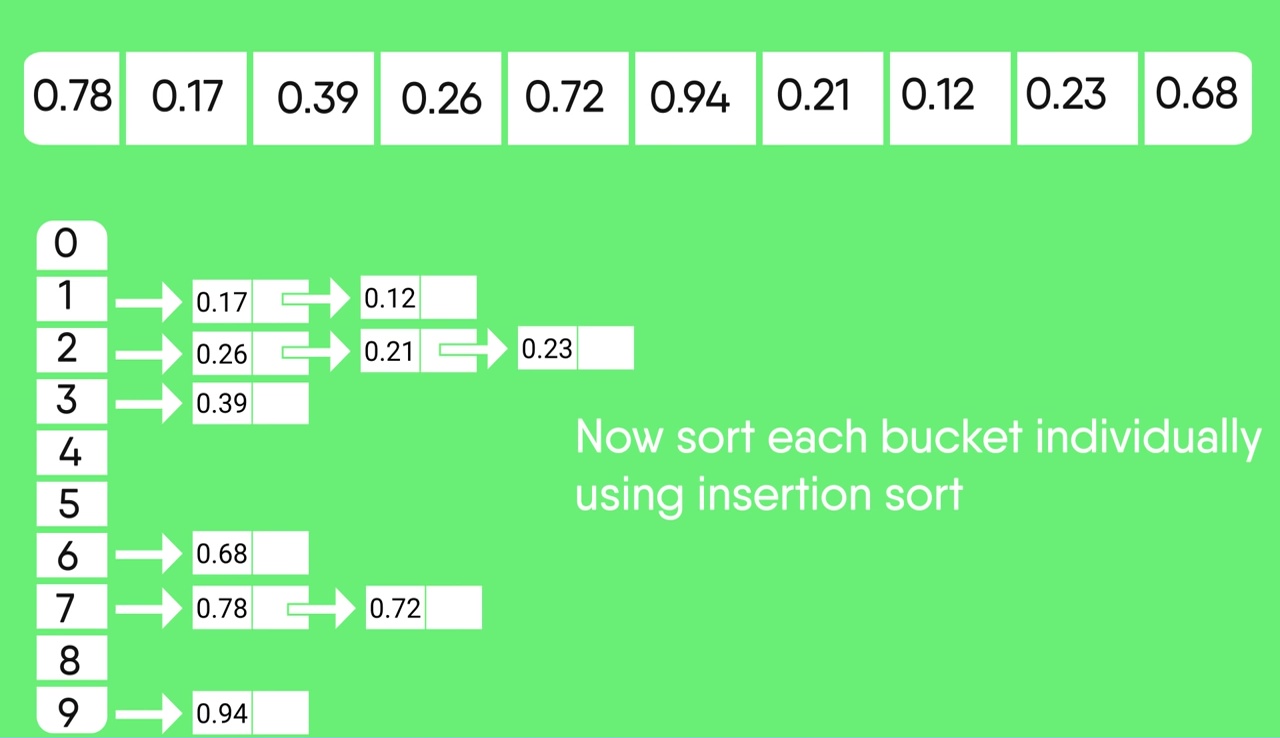
**Notes**

* Uses insertion sort as sub routine.
* It will work very well in the case like sort all double between 0 and 1
* If the data is very evenly distributed then bucket sort helps.
* The right size of buckets has to be chosen.
* It will perform O(n^2) when all elements are in one bucket.
* That is why knowing the minimum and maximum is important

**Analysis of Sorting Algorithm**

* k is number of buckets

|  |  |  |  |
| --- | --- | --- | --- |
| Best Case Time | Worst Case Time | Average Case Time | Space Complexity |
| O(n + k) | O(n + k) | O(n2) | O(1) |



Radix Sort

**How it works**

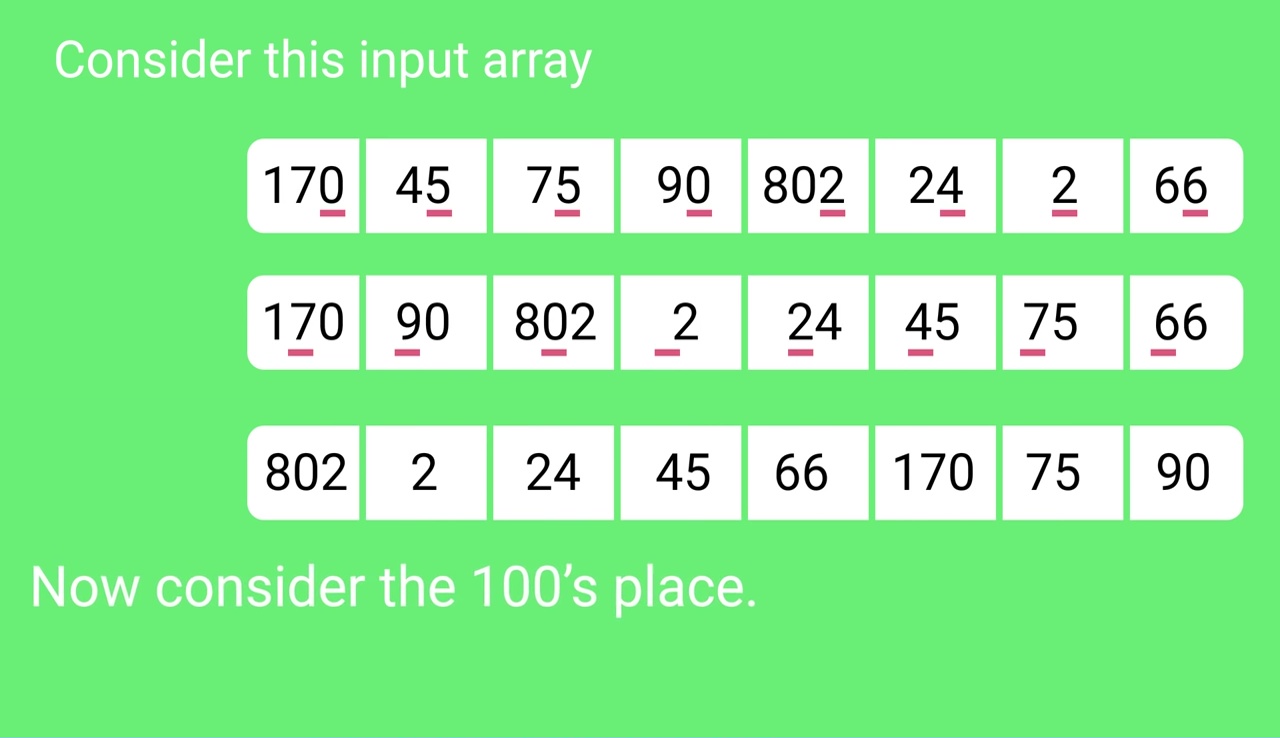
* First find how many digits are there in the biggest number in the sort.
* Then use a for loop to send each digit one by one to that function starting with the least digit.
* The function looks at that specific position using division and modulus(%).
* At each position look at the numbers and solve using counting sort.

**Notes**

* Uses counting sort as sub routine.

**Analysis of Sorting Algorithm**

|  |  |
| --- | --- |
| Time Complexity | Space Complexity |
| O(nk) n is the number of elements and k is the number of digits | O(1) |



Counting Sort

**How it works**

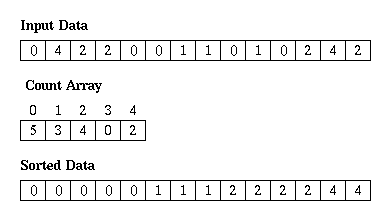
* First find the largest term of the array. Let's say that term is X.
* Now make an array from 0 to X. Let's call it count array.
* For every term that occurs, increment that specific index in count array by 1. Example if you see term 8 then count[8]++.
* Now go through the count array and keep adding the n-1th term in count array to nth term.
* Now the count array actually carries the index of the terms.
* Now go through each term in the original array and see its index in the count array and put that term in the new array at that specific index.

**Example**

* [4, 1, 6, 3, 4]
* [0, 1, 1, 2, 4, 4, 5] - Count Array
* [,,,,4,]
* [1,,,4,]
* [1,,,4,6]
* [1,3,,4,6]
* [1,3,4,4,6]

**Analysis of Sorting Algorithm**

|  |  |
| --- | --- |
| Time Complexity | Space Complexity |
| O(n + k) n is the number of elements and k is the range of input | O(1) |



Binary Search Tree

**How it works**

* The root is the top of the tree.
* Any element smaller than the root will go the left.
* Any element greater than the root will go the right.

**Time complexity of searching a binary tree in terms of h where h is the height of the tree:**

|  |  |  |
| --- | --- | --- |
| Best Case Time | Worst Case Time | Average Case Time |
| O(1) | O(h) | O(h) |

**The two cases are as follows:**

**When the tree is balanced**

|  |  |
| --- | --- |
| Worst Case Time | Average Case Time |
| O(log2n) | O(log2n) |

**When the tree is not balanced and is extremely imbalanced i.e. it only has right terms**

|  |  |
| --- | --- |
| Worst Case Time | Average Case Time |
| O(log2n) | O(log2n) |

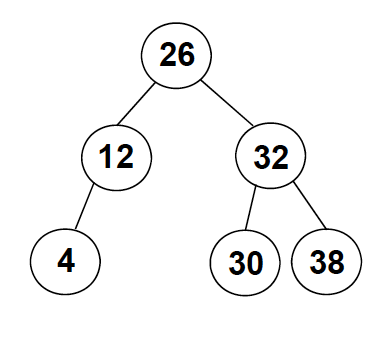
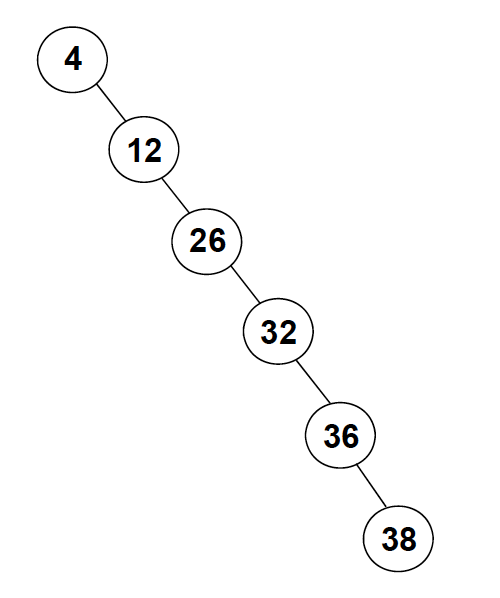
****

Figure 1: Extremely Imbalanced Tree

Figure 1: Balanced Tree

Heap Sort

**How it works**

* First change a normal array to a tree. Here the first element becomes the node. The second element becomes the left node. The third element becomes the right node. The fourth element becomes the left node of the left node. And so on.
* Now heapify the tree.
* This means if in any part of the tree, if node is smaller than its children then swap the node with the bigger child.
* The swapping is done in both the tree and the array. The swapping in array is done with the following rule:
  + Lets say the index of the node is I.
  + If you have to swap with the left child then swap arr[I] with arr[2\*I + 1]
  + If you want to swap with the right child then swap arr[I] with arr[2+I + 2]
* Now you have the largest node at the top so you can delete this node. By deleting, you shift this specific node to the end of the array.
* Now that you delete the node, you have to replace with the rightmost node of the left subtree which will be the last element of the array. So its like swapping the largest element and last element of the array.
* Keep doing this until you have a tree of size 1.

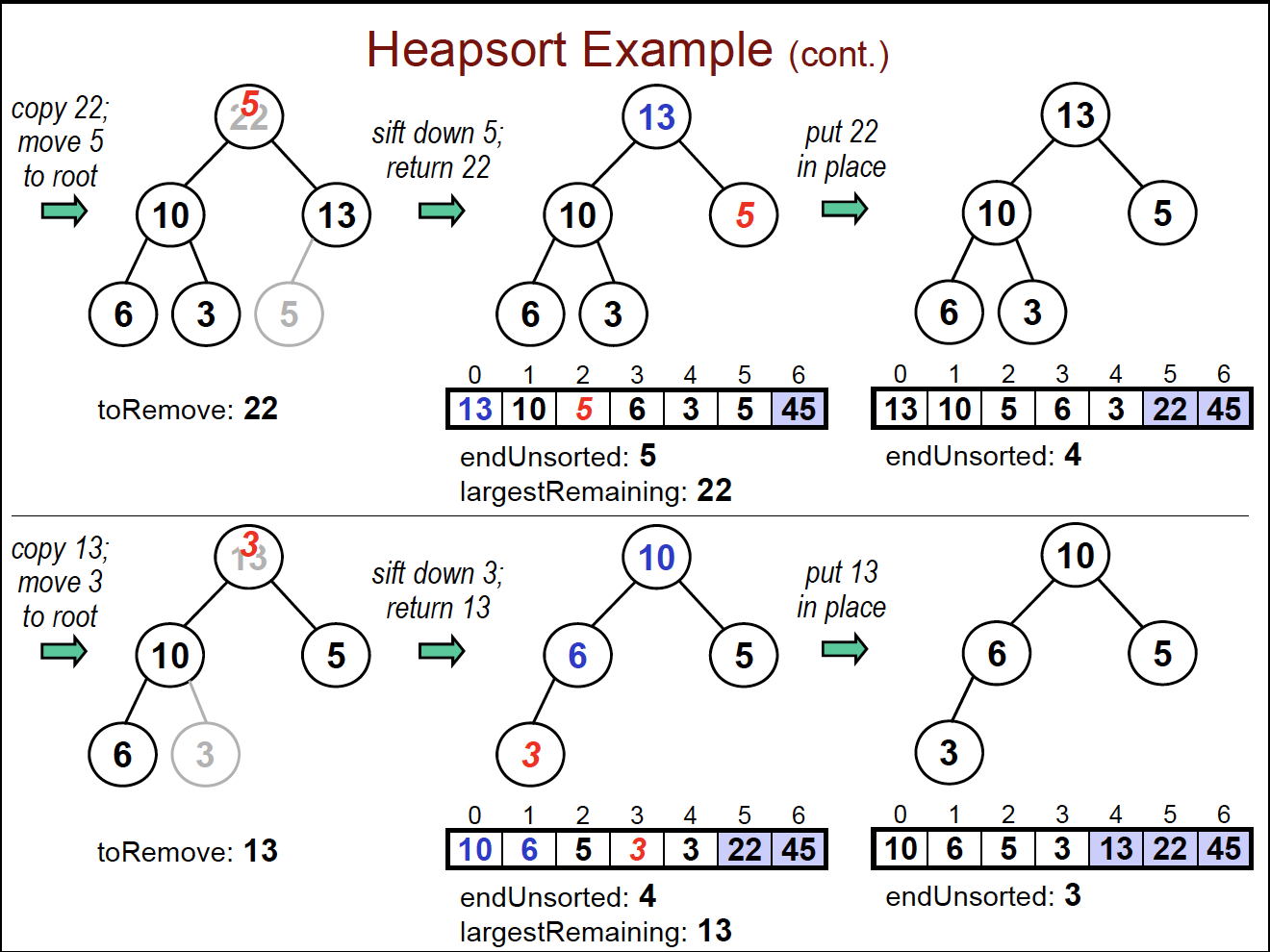
**Analysis of Sorting Algorithm**

|  |  |  |  |
| --- | --- | --- | --- |
| Best Case Time | Worst Case Time | Average Case Time | Space Complexity |
| O(n log n) | O(n log n) | O(n log n) | O(1) |

**Notes:**

* This is the fastest in case of worst case and is equivalent to merge sort but has better space complexity.
* When converting random array to heap:
  + If the node in array is a[i].
  + Its left child is a[2\*I + 1].
  + Its right child is a[2\*I + 2]
  + Its parent is a[(i-1)/2] or a[(i-2)/2], whichever gives a while number.

**Example of Sorting Algorithm**



Hash tables

**How it works**

* Hash table is basically a table(mostly an array) in which you can map depending on different characteristics of the information that are going to fill the table. The characteristic used are called keys.
* Example: a table that will have English words then the key can be the first character. Ant can go the first box, bear in the second and so on.
* Separate Chaining:
  + When multiple items are assigned to the same bucket(position) on the table then they form a chain. This chain can either be a linked list or an array though linked list is preferred. In case of array, if they are not filled will waste memory and if filled then will cause overflow. Linked list on the other hand only use up additional memory but in most cases are more efficient.
* Open Addressing:
  + When the position assigned is occupied, find another position in the same table.
  + Finding this position is called probing. Probing will also be used to find the elements.
  + The three types of probing are:
    - Linear Probing: If a space is not find go the next one. This is a problem when it find a whole cluster of filled positions.
    - Quadratic Probing: If a space is not filled, go to the next, then 4th, then 9th, then 16th and so on. The problem is it may fail to find an existing open space
    - Double Hashing: When it uses two codes, 1st for finding the initial position and the next for increments. 1st depending on the initial letter of the string. Increments depending on the number of letters in the String.
* If you removing an element from its position then you have to change the value of the position from empty to removed. Hence the three types of positions can be occupied, empty and removed.
* Elements can be inserted in both empty and removed positions.
* For Inserting, when probing, when we encounter an empty position then we stop but not when we encounter a removed position. For more details go to page 217.

**Analysis of Sorting Algorithm**

|  |  |
| --- | --- |
| Best Case Search and Insertion | Worst Case Search and Insertion |
| O(1) | Open Addressing: **O(m)** where m is the size of the has tables  Separate Chaining: **O(n)** where n is the number of keys in that specific bucket where you are searching or inserting. |

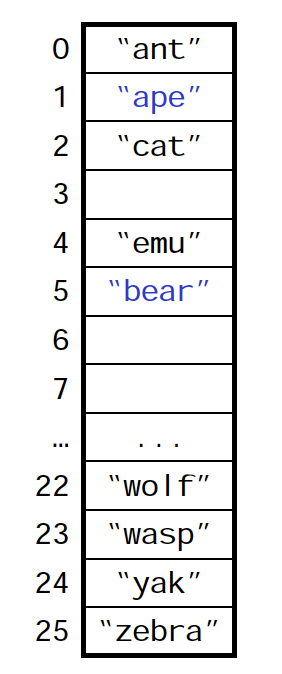
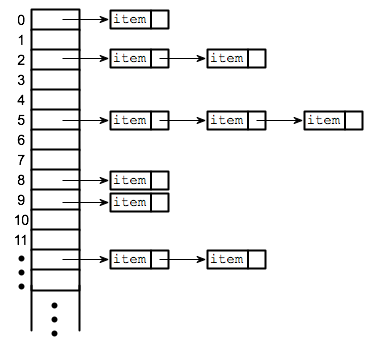
****

Figure 3: Separate Chaining

Figure 4: Open Addressing

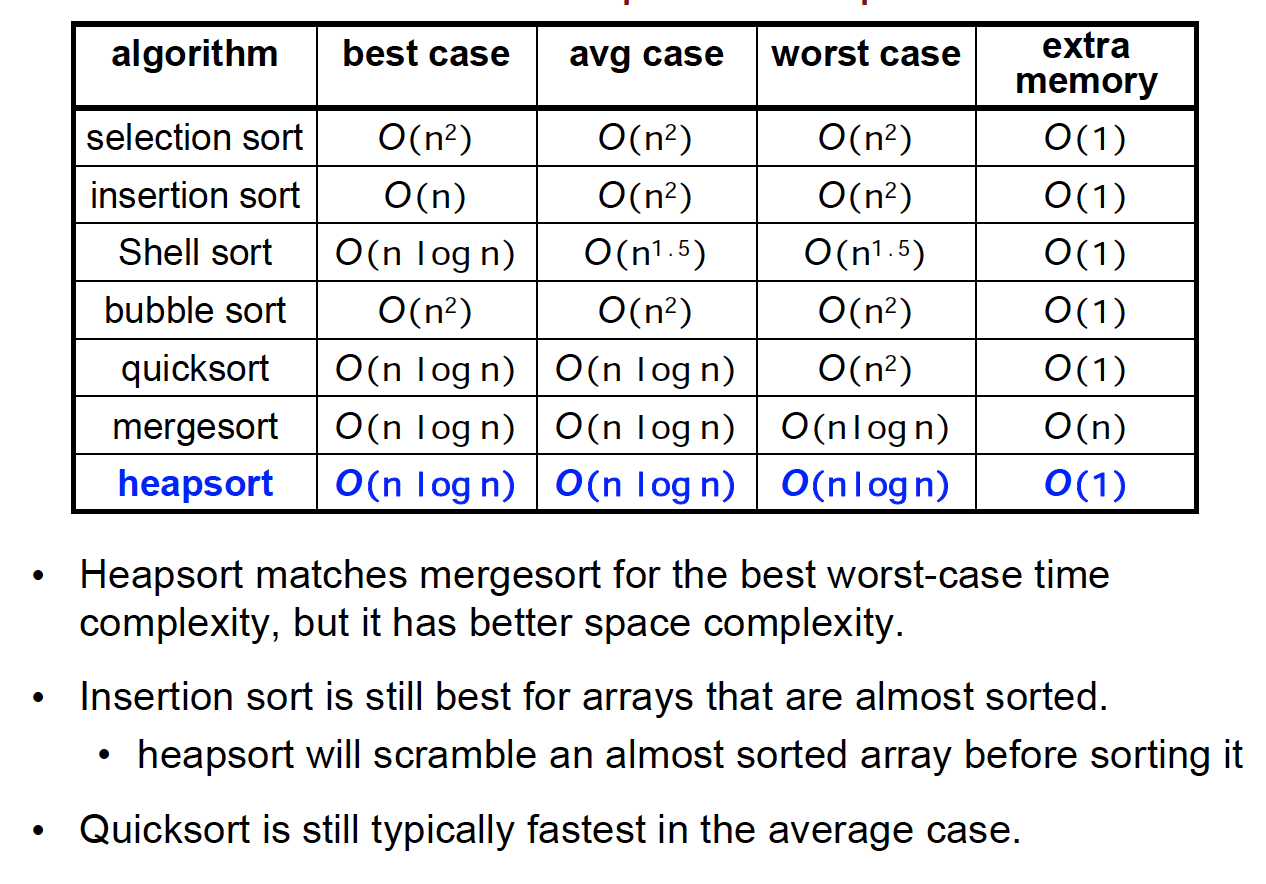
**Notes:**

* With good choice of hash function and table size. Complexity for separate chaining goes to O(log n).
* Bigger tables have better performance but they use more memory.

## Topological Sorting

* Have a hashset that contains all the nodes that are visited.
* Basically, keep going down every node. When you reach a node that has no other adjacent nodes that are not there in the hashset then add it to a stack.
* At then end the hashset will be topologically sorted.
* Following is the pseudocode:
  + Create a hashset and stack.
  + Now go to every node in graph. Only go to the node that are not in set.
  + Now when you accessing node, add it to the set.
  + When you accessing the node, access all its adjacent that are not in the set.
  + After accessing all the adjacent nodes.
  + Add the current accessing node to the stack.

All Sorting Algorithm



Hash Table

* Best Case:
  + **O(1)**
* Worst Case:
  + Open Addressing: **O(m)** where m is the size of the has tables.
  + Separate Chaining: **O(n)** where n is the number of keys in that specific bucket where you are searching or inserting.
  + With good choice of hash function and table size. Complexity for separate chaining goes to **O(log n**).
* Bigger tables have better performance but they use more memory.