Bubble Sort

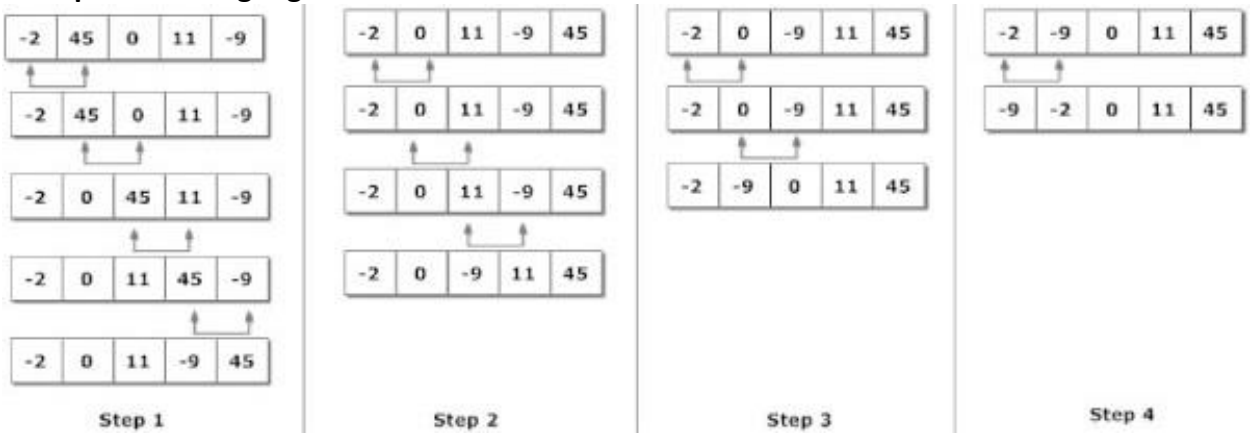
How it works

- 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
$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)

Example of Sorting Algorithm



Notes:

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

$$x! > 2^x > x^2 > x^{1.5} > x \log(x) > x > \log x.$$

- Always pass the whole array in whatever function you use
- Insertion Sort best when array almost sorted. Avg. case, quick sort quickest even though Merge and Heap are same. Merge & Heap best worst-case but heap better has space.

Selection Sort

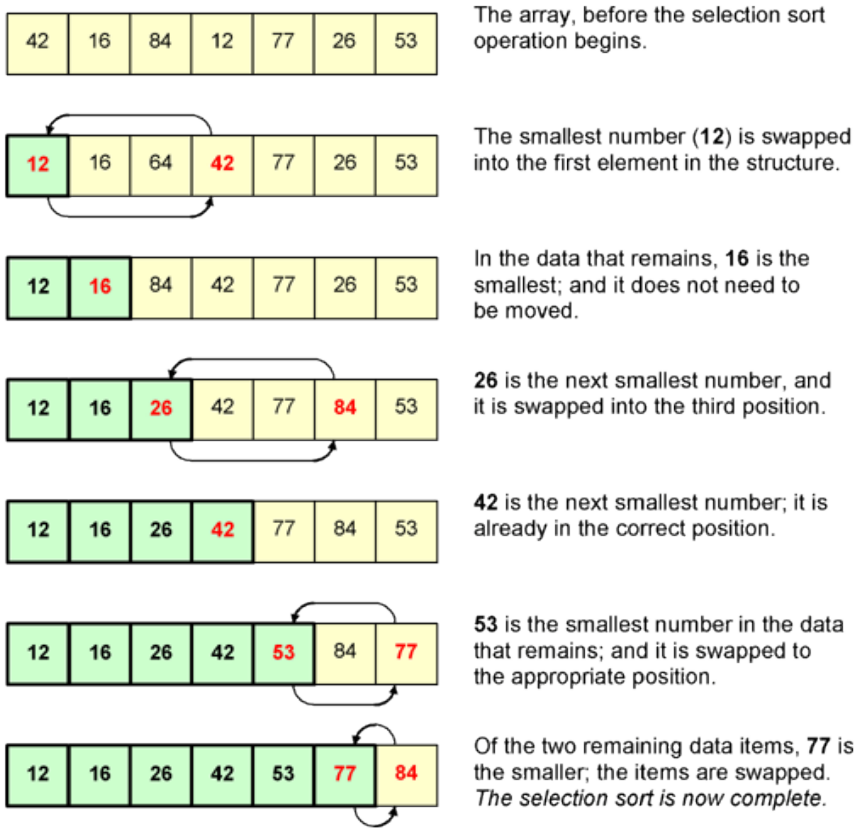
How it works

- 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
$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)

Example of Sorting Algorithm



Notes:

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

$$x! > 2^x > x^2 > x^{1.5} > x \log(x) > x > \log x.$$

- Always pass the whole array in whatever function you use
- Insertion Sort best when array almost sorted. Avg. case, quick sort quickest even though Merge and Heap are same. Merge & Heap best worst-case but heap better has space.

Insertion Sort

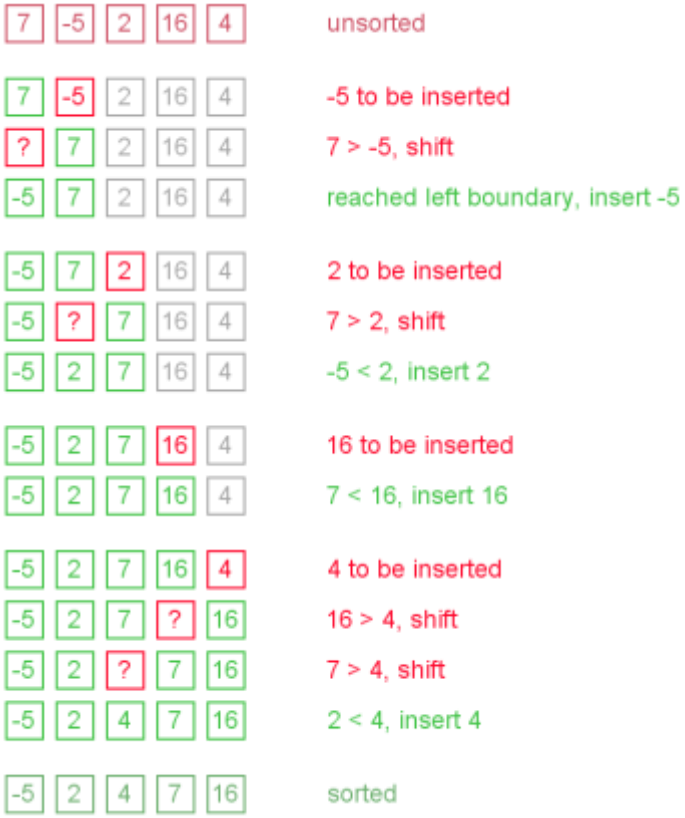
How it works

- 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)	$O(n^2)$	$O(n^2)$	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.

$$x! > 2^x > x^2 > x^{1.5} > x \log(x) > x > \log x.$$

- Always pass the whole array in whatever function you use
- Insertion Sort best when array almost sorted. Avg. case, quick sort quickest even though Merge and Heap are same. Merge & Heap best worst-case but heap better has space.

Shell Sort

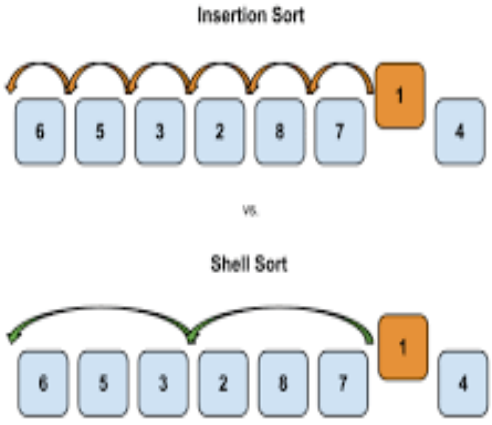
How it works

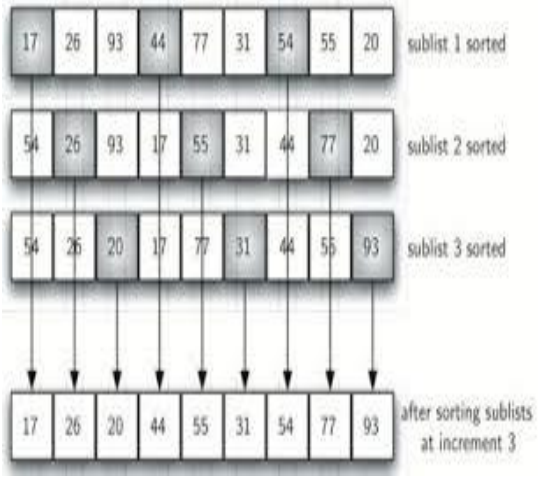
- Shell sort uses insertion sort on subarrays.
- The subarrays are in the same array but have a greater increment than 1.
- Lets say the increment is 3. You start at position 1, and see what values are there in 4, 7, 10 and so on and use insertion sort on this.
- Then you look at position 2, and see what values are there in 5, 8, 11 and so on and use insertion sort on that.
- Then you look at position 3 and this then continues at position 4, 5, 6 etc.
- Then start getting the increments down until they are 1.
- The increments should not be a multiple of each other, otherwise the elements that are sorted in one pass will be sorted again in the next pass.
- Prefer using increments that are one less than power of 2.
- **$2^k - 1$: 63, 31, 15, 7, 3, 1**

Analysis of Sorting Algorithm

Best Case Time	Worst Case Time	Average Case Time	Space Complexity
O(n log n)	$O(n^{1.5})$	$O(n^{1.5})$	O(1)

Example of Sorting Algorithm





Notes:

- This improves insertion sort.
- This takes advantage of the fact that insertion sort is fast when an array is almost sorted.
- This eliminates the disadvantage of insertion sort that if an element is away from its final position, many small moves are required to put it back.
- This allows larger moves to allow elements to reach their final place.

$$x! > 2^x > x^2 > x^{1.5} > x \log(x) > x > \log x.$$

- Always pass the whole array in whatever function you use
- Insertion Sort best when array almost sorted. Avg. case, quick sort quickest even though Merge and Heap are same. Merge & Heap best worst-case but heap better has space.

Quick Sort

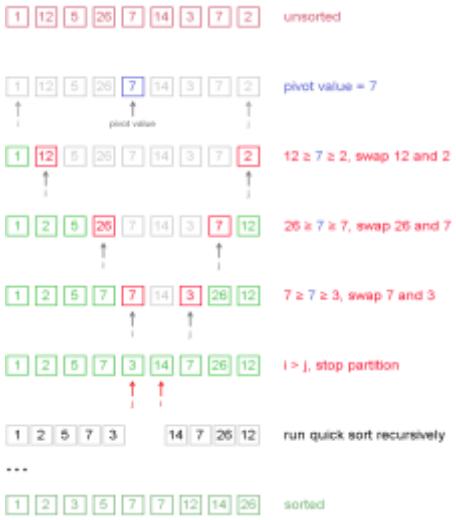
How it works

- 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 indices is greater than pivot then exchange it with the value at J indices which is higher than pivot.
- This keeps going on until I and J cross.
- The 2 subarrays then divide further into 4 subarrays.
- 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)	$O(n^2)$	O(1)

Example of Sorting Algorithm



Notes:

- Log n grows much slower than n and hence this is better.
- Worst case($O(n^2)$) is when the pivot is either the smallest or the biggest element.

$$x! > 2^x > x^2 > x^{1.5} > x \log(x) > x > \log x.$$

- Always pass the whole array in whatever function you use
- Insertion Sort best when array almost sorted. Avg. case, quick sort quickest even though Merge and Heap are same. Merge & Heap best worst-case but heap better has space.

Merge Sort

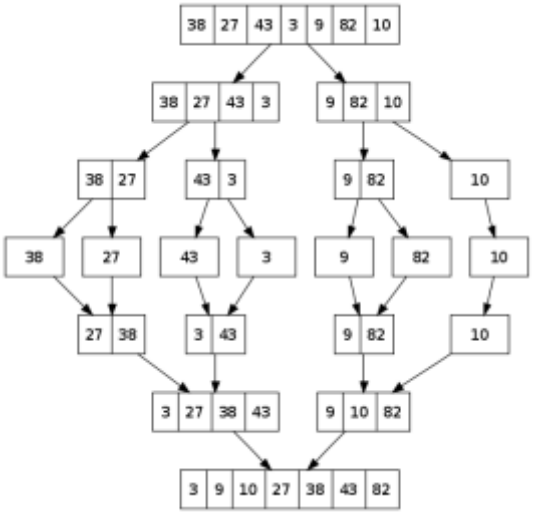
How it works

- 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

$$x! > 2^x > x^2 > x^{1.5} > x \log(x) > x > \log x.$$

- Always pass the whole array in whatever function you use
- Insertion Sort best when array almost sorted. Avg. case, quick sort quickest even though Merge and Heap are same. Merge & Heap best worst-case but heap better has space.

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
$O(\log_2 n)$	$O(\log_2 n)$

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

Worst Case Time	Average Case Time
$O(\log_2 n)$	$O(\log_2 n)$

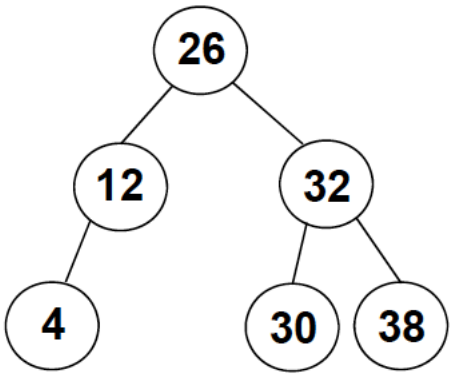


Figure 2: Balanced Tree

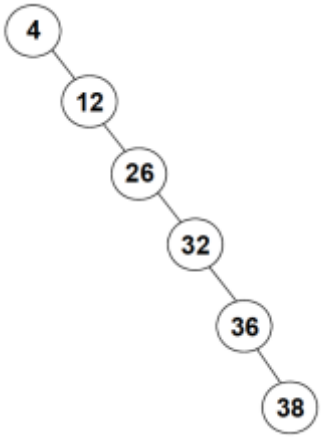


Figure 1: Extremely Imbalanced Tree

$$x! > 2^x > x^2 > x^{1.5} > x \log(x) > x > \log x.$$

- Always pass the whole array in whatever function you use
- Insertion Sort best when array almost sorted. Avg. case, quick sort quickest even though Merge and Heap are same. Merge & Heap best worst-case but heap better has space.

Heap Sort

How it works

- First change a normal array to a heap. A heap is in which the parent is always larger than the child.
- You first check the parent of last term and sift it down.
- In sifting you check the child of the parent and if the child is larger and whichever child is larger takes its place.
- You keep checking all the parents in the array and keep sifting it down.
- After this you will have the largest term as the root.
- Keep the root at the end of the array.
- Repeat everything leaving out the last term that you have just sorted.
- The array will start getting sorted from the end.

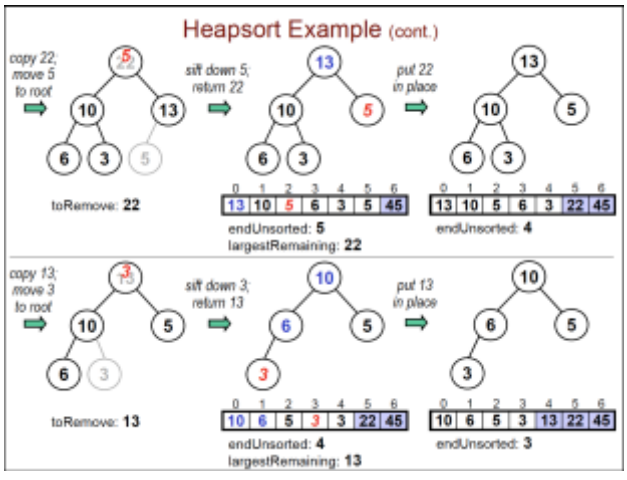
Analysis of Sorting Algorithm

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



$x! > 2^x > x^2 > x^{1.5} > x \log(x) > x > \log x.$

- Always pass the whole array in whatever function you use
- Insertion Sort best when array almost sorted. Avg. case, quick sort quickest even though Merge and Heap are same. Merge & Heap best worst-case but heap better has space.

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 4^{th}, then 9^{th}, then 16^{th} and so on. The problem is it may fail to find an existing open space
        - Double Hashing: When it uses two codes, 1^{st} for finding the initial position and the next for increments. 1^{st} 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.

$$x! > 2^x > x^2 > x^{1.5} > x \log(x) > x > \log x.$$

- Always pass the whole array in whatever function you use
- Insertion Sort best when array almost sorted. Avg. case, quick sort quickest even though Merge and Heap are same. Merge & Heap best worst-case but heap better has space.

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.

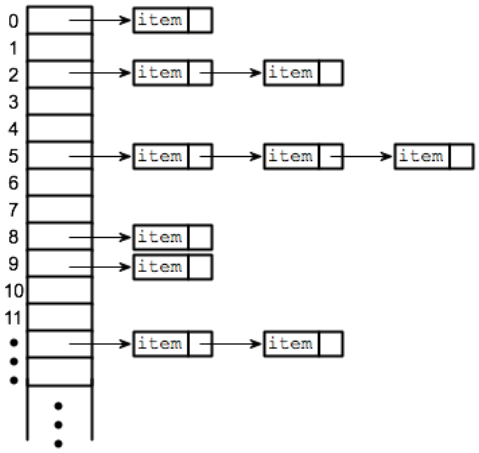


Figure 3: Separate Chaining

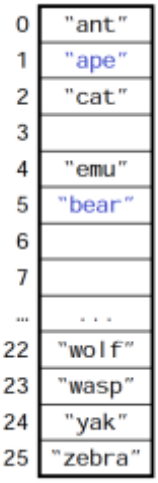


Figure 4: Open Addressing

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.

$$x! > 2^x > x^2 > x^{1.5} > x \log(x) > x > \log x.$$

- Always pass the whole array in whatever function you use
- Insertion Sort best when array almost sorted. Avg. case, quick sort quickest even though Merge and Heap are same. Merge & Heap best worst-case but heap better has space.

All Sorting Algorithm

algorithm	best case	avg case	worst case	extra memory
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell sort	$O(n \log n)$	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(1)$
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

- Heapsort matches mergesort for the best worst-case time complexity, but it has better space complexity.
- Insertion sort is still best for arrays that are almost sorted.
  - heapsort will scramble an almost sorted array before sorting it
- Quicksort is still typically fastest in the average case.

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.

$$x! > 2^x > x^2 > x^{1.5} > x \log(x) > x > \log x.$$

- Always pass the whole array in whatever function you use
- Insertion Sort best when array almost sorted. Avg. case, quick sort quickest even though Merge and Heap are same. Merge & Heap best worst-case but heap better has space.