**Dynamic Programming**

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial. For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear. This technique is called memorization.

The idea is that whenever we compute a fibonacci number we put it in a dictionary. And then when we need to compute another fibonacci number we check if it is already in the dictionary.

**Dynamic programming or DP** = Recursion + Memorization

**Sorting Algorithms**

**Bubble sort**

In the bubble sort, we start from the beginning, and as we move on over the array we check a[i] with a[i+1] if a[i] > a[i+1] then we swap them. So the time complexity is **O()** .

In this algorithm, the highest number will bubble its way to the right with each iteration. After each iteration our array’s length will be 1 unit less than the last iteration’s array’s length.

**Selection sort**

In this algoritm, we keep track of the following variables: the current item and the minimum. We start by setting the current minimum to the first element of the array and the current item to the second element of the array. Once we find an element that is less than our current minimum we set the current minimum to the current item and we move on and when we finish iteration we swap the current minimum with the element at the beginning of the array. After first iteration we are gonna have one item in our sorted partition. After this we set our current minimum to the second element in the array and the current item to the third element of the array and the process continues. The time complexity is obviously **O()** .

**Insertion sort**

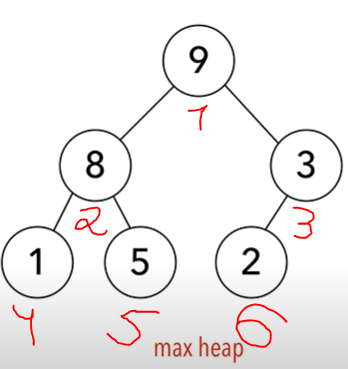
In this algorithm, we start from the beginning of the array and as we go forward we check the right and the left element. If arr[i]>arr[i+1] then we swap arr[i] with arr[i+1]. After this we check the swapped arr[i] with the element that it on the left of arr[i] that is to say arr[i-1] and this process continues. So this is also **O()** . Nevertheless, this algorithm is better than bubble sort and selection sort because it sorts in place meaning that it doesn’t just move on after swapping a[i] and a[i+1] it makes sure that the left side is always sorted.

**Heap sort**

In this algorithm, we have to be familiar with a few terminologies.

Heap – heap is just an ordered binary tree.

Max heap – max heap has a restriction that the values of parent nodes are greater than the values of the child nodes.

 **Max Heap example**

We make use of a few functions when running heap sort:

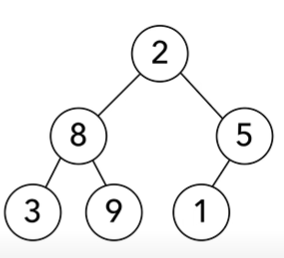
Build-max-heap: this function creates max heap from unsorted array

Heapify: similar to build-max-heap except for the fact that we assume that the part of the array is already sorted.

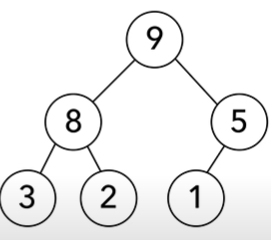
Suppose we have a following array 🡪



Our tree will look like this 🡪



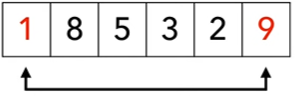
After this we call build-max-heap 🡪



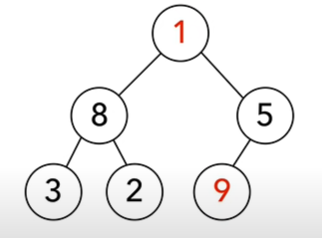
Then we rebuild our array. Our array will look like this🡪



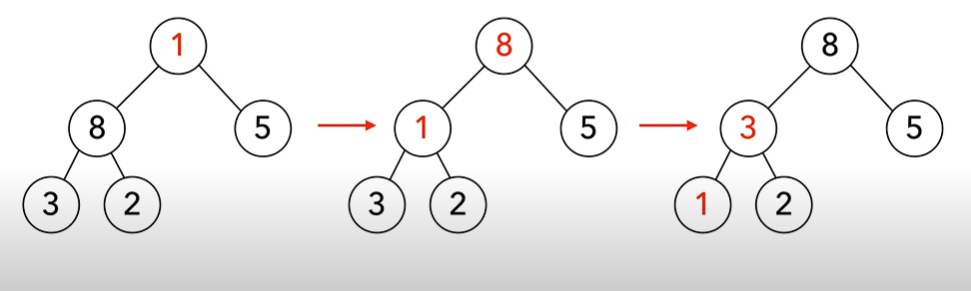
Now that root element has a value of 9, we know that 9 is the largest element in the array. We swap 9 with the item at the end of the array which is 1. Then our array 🡪



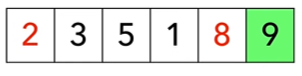
Our tree🡪



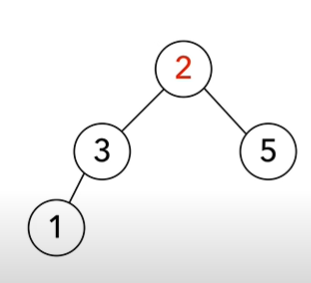
Finally, we remove 9 from the tree and consider it sorted. But now instead of having a heap we have a tree. This we call the heapify function! Since one part of our array is already sorted (last item).



Now we again have a heap. Now we swap 8 with the last item of the our unsorted part of the array 🡪



Then we remove 8 from the tree and place 2 in the root of our tree 🡪



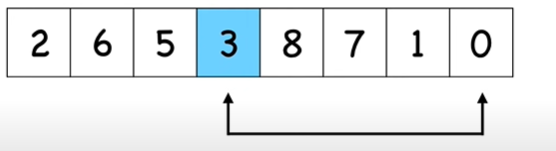
Then again we call heapify and the process continues. The time complexity for this alrgorithm is **O(n)**.

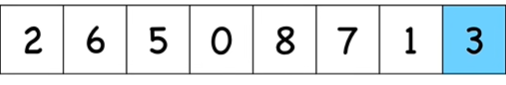
**Quick sort**

In this algorithm, we have a concept of a pivot. A pivot is simply one of the items in the array that meets the following 3 conditions after we sort the array. 🡪

1. Pivot is in its correct position when the array is sorted
2. Items to the left of the pivot are smaller than the pivot
3. Items to the right of the pivot are greater than the pivot.

First we need to choose a pivot. The best pivot is the median of the array. We choose the pivot by comparing the first, the last, and the middle elements of the array and choose the median one. So we choose the pivot then we move it the the end of the array 🡪

 our pivot is 3 in this example

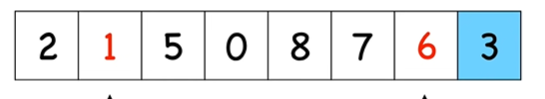


Next we are gonna look for 2 items

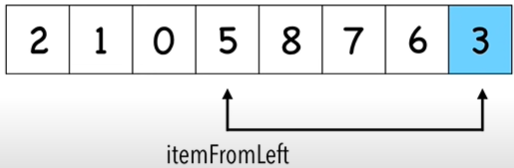
1. itemFromLeft that is larger than the pivot so that we put it in the right
2. itemFromRight that is less than our pivot so that we swap it with the itemFromLeft.



Once we find the the right elements we swap them 🡪



We continue the process but this time 5 is gonna be itemFromLeft and 0 itemFromRight. Then again we swap the two. When we see that itemFromLeft has a greater index than itemFromRight we know that we are done. Then we swap item from left with our pivot 🡪





Now 3 our pivot is in its correct spot. Then we do the same process for the right and left parts of our array.

The time complexity of the average case is **O(n)**.

**Merge sort**

In this sorting algorithm, we are gonna split our problem into subproblems, meaning that we are going to use divide and conquer approach. Once, we have the base case which is the case when we have one element in the array we are gonna start comparing and merging the arrays. The time complexity for merging is O(n) and the division is O(logn) so the time complexity of the merge sort is O(nlogn).

**Linked List**

Linked lists are preferrable when there is no requirement for storing data in a specific order and when there are fast removals and additions to to beginnig or end of the Linked List since now we don’t have to shift the entire list when removing data. We just will change the references of 2 elements in the linked list. But inserting or removing an element from the middle or somewhere else without having the node (which has to be deleted or to which a new node will be inserted before) is slow and will be O(n). Also if there have to be a lot of iterations then a simple list would suffice.

Also adding to the end of the linked list (scaling) it is very fast

So if we just have an index of some node in a linked list before which we want to insert a new node then it is slow. Because we will need to iterate through it. But if we have the node then it is O(1). So it really depends on the situation.

It is very scalable. But lookup will not be fast (O(n)).

**Stack**

Stacks are better when we want to undo what we have just done. Last In First out. Ctrl z works with Stacks. Stacks are ideal for undo, call stacks (like in in threads) etc. (Lifo)

**Queue**

This data structure is preferred when we need to process the longest waiting item (Fifo).

Stacks and Queues do not support direct lookup because there is no point. We can just enumerate over them. They only support peek operation to see the latest (if stack) or the longest item (if queue) in the collection.