DARC

Growth functions
A simple mathematical function is used to denote the complexity of algorithms.
- Primary parameter N - size of the problem - affects the running time most significant.
- N might be
  - the size of a file to be sorted or searched
  - the number of characters in a text string or
  - some other measure of the size of the problem being considered.
- Usually N is directly proportional to the size of the data set being processed.

Dimension of the problem
Our goal is to express the resource requirements (mostly time but also space) in terms of N, using mathematical formulas.
- We are only interested in large values of N
- The most common functions that we will need are:
  - 1
  - log(x)
  - x,
  - x-log(x)
  - x^2
  - x^3
  - …
- Using these functions we can make classification for all algorithms

Growth function: N
Linear running time.
- When a limited amount of processing is done, on each input element the running time of the program is linear. This running rime occurs e.g. in a method where we search a value in an array or list.

Growth function: 1
Constant running time
- Most instructions of most programs are executed once or at most only a few times. If all the instructions have this property, we say that the program's running time is constant.
- Doubling the input length N hardly influences the running time.

Growth function: log(N)
Logarithmic running time
- When the running time of a program is logarithmic then the running time grows slower that any positive power of the N.
- We have seen program with a logarithmic running time: Binary search

Growth function: N * log(N)
Running time proportional to N * log(N)
- The N * log(N) running time can arise when algorithms solve a problem by breaking it into smaller subproblems solving independently and then combining the solutions.
- When N doubles, the running time grows a bit more but always slows than N to a power above 1.
- Example: Quicksort & Heapsort

Big O Notation and Time complexity

Linear time complexity - when the time increases linearly as the input grows. (O(N))
Constant time complexity - when the time does not change no matter the input. (O(1))
Quadratic time complexity - when time increases quadratically together with increasing input. (O(N^2))

Linear search (O(n))

Linear search is a simple searching algorithm that looks for a target value in a list sequentially. It checks each element of the list until a match is found or the entire list has been searched.

Code:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

Binary search O(log N)

Binary search is a more efficient algorithm, but it requires that the data set is sorted. It repeatedly divides the search interval in half and compares the middle element with the target value. Depending on the comparison, the search continues in the left or right half until the element is found or the search interval becomes empty.

Code:

```
function binarySearch(arr, target):
    left = 0
    right = length of arr - 1
    while left <= right:
        mid = (left + right) / 2
        if arr[mid] equals target:
            return mid  // element found at index mid
        else if arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1  // element not found
```

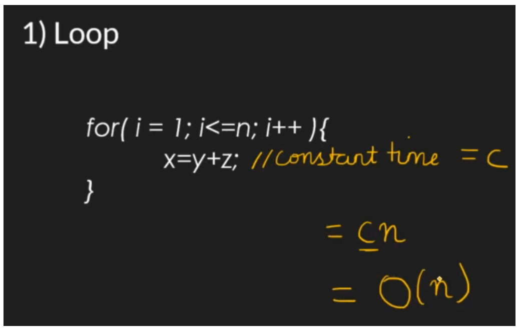
Time complexity is analysed for:

- Very large input size
- Worst case scenario

Example:

T(n) = 2n^2 + 3n + 1
      = O(n^2)

Time Complexity of a single loop:



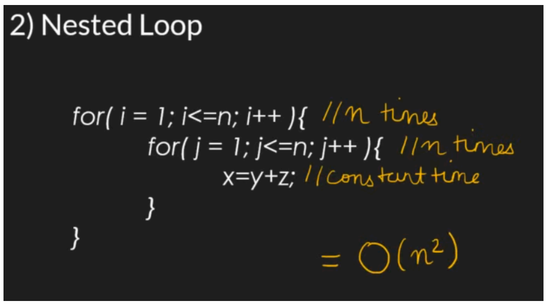
1) Loop

```
for( i = 1; i<=n; i++ ){
        x=y+z; //constant time  = c
}
```

$= cn$

$= O(n)$

Nested loop:

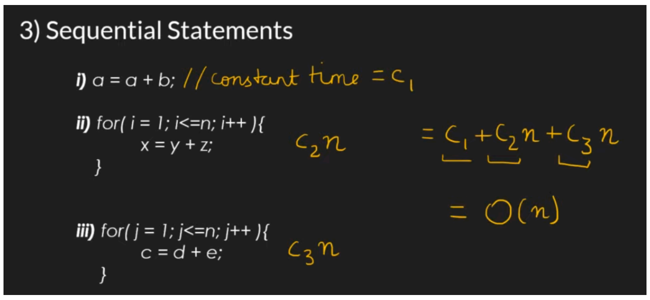


2) Nested Loop

```
for( i = 1; i<=n; i++ ){ //n times
        for( j = 1; j<=n; j++ ){ //n times
                x=y+z; //constant time
        }
}
```

$= O(n^2)$

Sequential statements:



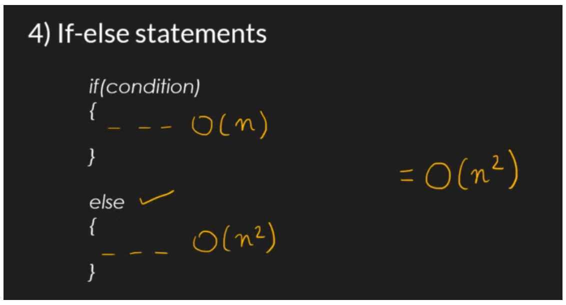
3) Sequential Statements

```
i) a = a + b; //constant time = c1

ii) for( i = 1; i<=n; i++ ){        c2 n
        x = y + z;
    }

iii) for( j = 1; j<=n; j++ ){       c3 n
        c = d + e;
     }
```

$= \underbrace{c_1} + \underbrace{c_2 n} + \underbrace{c_3 n}$

$= O(n)$

If-else statements:

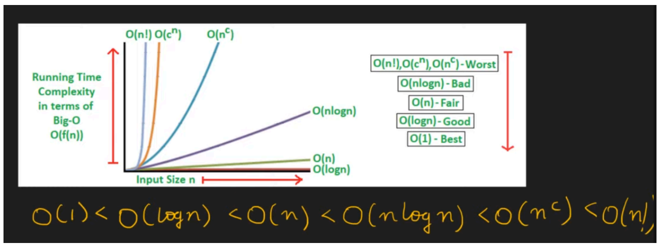


4) If-else statements

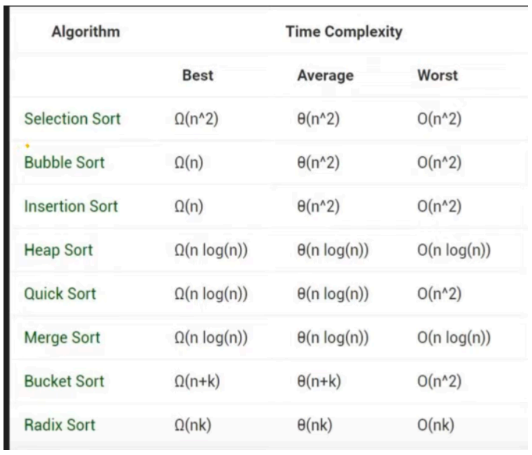
```
if(condition)
{ _ _ _ O(n)
}
else ✓
{
 _ _ _ O(n²)
}
```

$= O(n^2)$

Comparison of time complexities:



$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^c) < O(n!)$



Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	θ(nk)	O(nk)

Time-space trade-off and efficiency:
- Trade off between memory use and runtime performance
- Space efficiency and time efficiency reach at two opposite ends
- The more time efficiency you have, the less space efficiency you have and vice versa.

Example - Merge sort is very fast but requires a lot of space to do the operations. On the other hand Bubble sort is slow, but requires minimum space.

Sorters

Insertion - efficient on small data chunks but bad on large number of data. Insertion has a good performance on almost-sorted data.

Insertion sort is a simple and intuitive comparison-based sorting algorithm. It builds the sorted portion of the list one element at a time by iteratively taking an element from the unsorted portion and inserting it into its correct position within the sorted portion. The algorithm starts with the first element as the initially sorted portion and gradually expands it until the entire list is sorted. Insertion sort is particularly efficient for small datasets or partially sorted lists. It has an average and worst-case time complexity of O(n^2), but its adaptive nature allows it to perform better on nearly sorted data with a time complexity closer to O(n). Insertion sort is often used in practice for its simplicity and effectiveness on small or partially ordered datasets.

Start on left, moving to the right. Examine each item and compare it to items on it's left. Insert the item in the correct position in the array. The array will form sorted and unsorted partitions. This algorithm takes the first unsorted partition's item and checks if current item is smaller than the item on it's left. If current item is smaller, it gets swapped with an item on it's left (Then this gets repeated until the item on the left is smaller), if not it stays in the same position.

Selection - efficient on small data chunks but bad on large number of data. Used when write performance is a limiting factor.

Selection sort is a straightforward comparison-based sorting algorithm. It divides the input list into two parts: a sorted portion on the left and an unsorted portion on the right. The algorithm repeatedly selects the smallest (or largest, depending on the desired order) element from the unsorted portion and swaps it with the first unsorted element, expanding the sorted portion. This process continues until the entire list is sorted. Selection sort has a time complexity of O(n^2) and is not suitable for large datasets due to its inefficiency, but its simplicity makes it easy to understand and implement.

Current minimum - current smallest number. In the beginning looking for a smaller current minimum (in this case the first element) and if a smaller item is found it is exchanged places with the current item. Then choosing next item and traversing to see if it is the smallest of the list. If a smaller one is found, that item is set to the current minimum and after traversing the list fully it is put next to the previous smallest item (exchanged places with the current item).

Bubble - Bubble sort is a straightforward comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the entire list is sorted. In each pass, the largest unsorted element "bubbles up" to its correct position at the end of the list. Bubble sort has a time complexity of O(n^2), making it inefficient for large datasets, but its simplicity and ease of implementation make it useful for educational purposes or small datasets. It is not commonly used in practice for large-scale sorting tasks due to its suboptimal performance compared to more advanced sorting algorithms.

How it works:
Takes two elements and compares their values, if left one is bigger, they get swapped.

Heap - Heap sort is a comparison-based sorting algorithm that uses a binary heap data structure to organise elements and efficiently extract the minimum (or maximum) element repeatedly. The algorithm consists of two main phases: heapify and sorting.

1. Heapify Phase:
    - Convert the input array into a binary heap, where the heap property is satisfied (either min-heap or max-heap). This ensures that the root element of the heap is the minimum (or maximum) element.
2. Sorting Phase:
    - Repeatedly extract the root element from the heap (which is the minimum or maximum) and swap it with the last unsorted element.
    - Reduce the heap size by one and restore the heap property.
    - Repeat these steps until the entire array is sorted.

Heap sort has a time complexity of O(n log n) in all cases, making it more efficient than some quadratic-time sorting algorithms. While not as popular as quicksort or mergesort for general-purpose sorting, heap sort is often used in situations where a stable sort is not a requirement, and in-place sorting with a consistent O(n log n) time complexity is desirable.

How it works:
Heap - ordered binary tree
maxHeap - parent > child
minHeap - parent < child

Functions to use:
BuildMaxHeap: creates max heap from unsorted array. Time complexity O(n)
*Heapify: similar to buildMaxHeap, but assumes part of array is already sorted. Time complexity O(log*n), called n-1 times*

1. *Create max heap*
2. *Remove largest item*
3. *Place item in sorted partition.*

Quick - Quicksort is a highly efficient, comparison-based sorting algorithm that follows the divide-and-conquer strategy. The algorithm works as follows:

1. Partitioning:
    - Choose a pivot element from the array.
    - Rearrange the elements in the array such that all elements less than the pivot are on its left, and all elements greater than the pivot are on its right.
    - The pivot is now in its final sorted position.
2. Recursion:
    - Recursively apply the above steps to the subarrays on the left and right of the pivot until the entire array is sorted.

Quicksort is known for its average-case time complexity of O(n log n), making it one of the fastest sorting algorithms in practice. However, its worst-case time complexity is O(n^2), which occurs when the pivot selection consistently results in poorly balanced partitions. Various strategies, such as randomising pivot selection or using the "median of three" method, can be employed to mitigate the risk of worst-case behaviour.
Quicksort is widely used in practice due to its efficiency, in-place nature (requiring minimal additional memory), and adaptability to different data distributions.

How it works:

- *In the final, sorted array the pivot has to meet 3 conditions - correct position in final, sorted array; Items to the left are smaller; items on the right are larger.*

- *Choosing pivot - popular way is median of three. At first choosing three items - first, middle and last items, then sorting them so that the middle item has middle value of those three. Then the pivot is swapped places with the last element.*
- *itemFromLeft - first item from the left that is larger than the pivot*
- *itemFromRight - first element from the right, that is smaller than the pivot.*
- *When these items are found, they are switched places so that the smaller element ends up at the left side of the array and the bigger one, at the right.*
- *When the index of itemFromLeft is bigger than index of itemFromRight, itemFromLeft gets switched up with the pivot. Both sides are then sorted individually and so on until the array is sorted.*

Merge - merge sort is a comparison-based sorting algorithm that follows the divide-and-conquer paradigm. The algorithm works as follows:

1. Divide:
   - Divide the unsorted array into two halves.
2. Conquer:
   - Recursively sort each half. This is done by applying the merge sort algorithm to each of the subarrays.
3. Merge:
   - Merge the sorted subarrays to produce a single sorted array.
   - This involves comparing elements from the two subarrays and placing them in the correct order in a new array.

The key idea behind merge sort is that it divides the sorting problem into smaller subproblems, solves them independently, and then combines the solutions in a way that ensures the final array is sorted. Merge sort has a consistent time complexity of O(n log n), making it more predictable than quicksort in terms of performance.

While merge sort is not as space-efficient as some in-place sorting algorithms because it requires additional space for the merging step, its stable nature (preserving the order of equal elements) and reliable performance make it a popular choice for sorting large datasets or linked lists.

How it works:
At the beginning divide the array in 2 parts and those parts again until only individual items are left. Then it puts it back together but sorting in every step. So individual items are taken as pairs and sorted between each other, then every two pairs are being merged into a group of 4 but sorted on the way.

Bucket - Bucket sort is a distribution-based sorting algorithm that divides an input array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm or by recursively applying the bucket sort algorithm. Finally, the sorted elements from all the buckets are concatenated to produce the fully sorted array.

The key steps in the bucket sort algorithm are as follows:

1. Distribute into Buckets:
   - Divide the input array into a number of equally spaced buckets.
   - Distribute the elements of the array into these buckets based on their values.
2. Sort Each Bucket:
   - Sort each individual bucket, either using another sorting algorithm or by recursively applying the bucket sort.
3. Concatenate Buckets:
   - Concatenate the sorted buckets to produce the final sorted array.

Bucket sort is particularly effective when the input data is uniformly distributed across a range. Its time complexity depends on the method used to sort each bucket and the number of buckets created. In the best case, where the data is evenly distributed among the buckets, the time complexity can approach O(n + n^2/k + k), where n is the number of elements and k is the number of buckets. However, in the worst case or when poorly chosen bucket sizes lead to skewed distributions, the time complexity may degrade.

Bucket sort is suitable for scenarios where the input data is distributed across a range and the range can be easily divided into buckets. It is not as widely used as some other sorting algorithms like quicksort or mergesort in general-purpose sorting.

Radix - Radix sort is a non-comparative sorting algorithm that works by distributing elements into buckets based on their individual digits or radix. The process is repeated for each digit, moving from the least significant digit (LSD) to the most significant digit (MSD) or vice versa, until the entire array is sorted.

The steps of the radix sort algorithm are as follows:

1. Distribute into Buckets:
   - Starting with the least significant digit (or most significant, depending on the implementation), distribute the elements into buckets based on that digit.
2. Collect Buckets:
   - Collect the elements from the buckets in order, creating a new array.
3. Repeat:
   - Repeat the process for each subsequent digit until the entire array is sorted.

Radix sort can be applied in two variants: LSD (Least Significant Digit) and MSD (Most Significant Digit). In LSD radix sort, the algorithm starts sorting from the rightmost digit, while in MSD radix sort, it starts from the leftmost digit.

Radix sort is often used with integers or fixed-size strings. Its time complexity is determined by the number of digits in the maximum number in the array. If k is the number of digits, the time complexity is O(kn), where n is the number of elements in the array. Radix sort is a stable sort, meaning that the relative order of equal elements is preserved.

While radix sort has linear time complexity in some cases, it may not be as efficient as comparison-based sorting algorithms like quicksort or mergesort for small datasets or in situations where the number of digits is large. However, it can be very effective for sorting large datasets with fixed-size keys.

Perfect & subset sums

Subset sum - finding if the set can sum up to a target sum. Example - if the set is 1,3,4,2,2, and the target is 6, the answer is 'yes'

Perfect sum - finding if the set can sum up to a target sum. Example - if the set is 1,3,4,2,2, and the target is 6, the answer is Subset 1: {1,3,2}; Subset 2:{4,2}. In perfect sum there should be no duplicates meaning the values in each subset should be distinct. Complexity - O(n * K) where n is the number of elements and K is the target sum.

Tree traversal

The process of traversing (visiting, accessing or processing) all nodes in a tree data structure. There are several method for tree traversal:

Inorder Traversal

- Traverse the left subtree
- Visit the root node
- Traverse the right subtree
- For binary search trees, this yields nodes in ascending order.

Preorder Traversal:

- Visit the root node.
- Traverse the left subtree.
- Traverse the right subtree.
- Useful for making a prefix expression (Polish notation) of an expression tree.

Postorder Traversal:

- Traverse the left subtree.
- Traverse the right subtree.
- Visit the root node.
- Useful for making a postfix expression (Reverse Polish notation) of an expression tree.

Level Order (Breadth-First Traversal):

- Traverse the tree level by level, from left to right.
- Visit all nodes at the current level before moving to the next level.
- Uses a queue data structure for efficient implementation.

X

Example:

```
    1
   / \
  2   3
 / \
4   5
```

Inorder Traversal: 4, 2, 5, 1, 3

Preorder Traversal: 1, 2, 4, 5, 3

Postorder Traversal: 4, 5, 2, 3, 1

Level Order: 1, 2, 3, 4, 5

Data Structures

Arrays:

Description: A collection of elements stored at contiguous memory locations, each identified by an index or a key.
Characteristics: Random access to elements, fixed size.

Linked Lists:

Description: A linear data structure where elements are stored in nodes, and each node points to the next node in the sequence.
Characteristics: Dynamic size, efficient insertion and deletion, sequential access.

Trees:

Description: A hierarchical data structure with a root node and subtrees of nodes, where each node has a value and child nodes.
Characteristics: Hierarchical structure, used for hierarchical relationships, search, and sorting.

Graphs:

Description: A collection of nodes connected by edges, representing relationships between entities.
Characteristics: Vertices (nodes) and edges, used for modeling complex relationships and networks.

Queues:

Description: A linear data structure where elements are added at the rear (enqueue) and removed from the front (dequeue).
Characteristics: Follows the First In, First Out (FIFO) principle, used for task scheduling and managing data in a sequential manner.

Stacks:

Description: A linear data structure where elements are added on top (push) and removed from the top (pop).
Characteristics: Follows the Last In, First Out (LIFO) principle, used for function call management, undo mechanisms, and parsing expressions.

Bag (Not necessary)

Removal of items is not possible. A bag is just a bag to put things in! An iterator can let you search in the bag
Also do not expect that an iterator gives you the items in the same order as they were added
You may, of course extend the ADT (Abstract Data Type), e.g. to keep the bag sorted. A name for this new ADT could be: SortedBag

Stability

Stability:

Stability is no issue at all if you use a composite Comparator

An array implementation of Selection Sort is either really inefficient or causes instability.

The issue of stability only comes into play when consecutive sort actions with different sort keys are executed.