Primitive Datatype: int, long, float, double, bool, char, byte

ADT: Abstract Data Type (Hiding details?) – what operation does and not how operation works???

OOP features: Encapsulation, Polymorphism and Inheritance

Goodness of Algorithm: Time complexity and Space complexity

Time complexity measures:

Experimental analysis – runtime measurement – advantages – easy to execute – drawbacks – limited input, hardware dependency, software dependency

Theoretical analysis – performed on description of algorithm – advantages – independent of hardware/software/OS and considers all inputs ( all primitive operations are considered constant, frequency of occurrence is calculated)

Types of statements (primitive operations) for theoretical analysis:-

Declarations, Assignment, Arithmetic operations, Comparison statements, Accessing elements, Calling functions, Returning functions

Sorting in while/for loops with index or end moving results in logN complexity.

Order of growth of functions can be constant, logarithmic, linear, N\*logN, quadratic (N^2), cubic (N^3), exponential (a^N) – latter orders have higher runtime and must be avoided

Asymptotic Analysis: asymptotic means approaching a value, limit of function growth at higher orders (very high N)

Best case: lower bound; Worst case: upper bound; Average case: inbetween; - Only ‘Worst case’ is used.

Asymptotic notations:

Big-Oh – O() measures upper bound, worst case – rate of growth relative to input size

Big-Omega – Ω() measures lower bound, best case -

Big-Theta – θ() measures tight bound, average case

Big-Oh notation: runtime function f(n) <= c\*g(n) where n >= 1(input size) and c > 1 (const) then f(n) <= O(g(n)). The smallest g(n) i.e., lowest upper bound is considered.

Big-Omega notation: runtime function f(n) >= c\*g(n) where n >= 1(input size) and c > 1 (const) then f(n) >= Ω(g(n)). The largest g(n) i.e., highest lower bound is considered.

Big-Theta notation: runtime function c2\*g(n) => f(n) >= c1\*g(n) where n >= 1(input size) and c > 1 (const) then f(n) == θ (g(n)).

Performance summary is dependent on worst case i.e., Big-Oh notation. If best case and worst case is same for an algorithm, Big-Theta notation is used.

For linear search. O(1) is best case, Ω(n) is worst case and θ(n) is average case.

Performance classification: constant – O(1), logarithmic – O(logN), linear – O(N), N\*logN – O(N\*logN), quadratic – O(N^2), cubic – O(N^3), exponential - O(a^N).

Space Complexity:- bytes consumed by the program

Primitive datatypes: bool – 1 byte; byte – 1 byte; char – 2 byte; int – 4 bytes; float – 4 bytes; long – 8 bytes; double – 8 bytes;

Array of primitive data-type – n\*(primitive data-type bytes)

Recursive functions – when a function calls itself upon its call. A base condition is established which limits how many times the function calls itself.

Iterative functions use loops i.e., worst case of O(n). Recursive functions call themselves and need recurrence relation for time complexity calculation.

Once the recurrence relation is calculated which has T(n) = c + T(n-1) form for eg., we can either use substitution (successive substitution or induction) method or master’s theorem (only works for few recurrence relations with conditions) to calculate total time complexity of recursive function.

Types of Recursions: Tail recursion, Head Recursion, Tree Recursion and Indirect Recursion.

Head and Tail Recursions call self before and after its statements respectively.

Tree Recursion calls self both before and after its statements and depending on base condition may execute quite differently to previous examples. The tree is the shape that the control traverses when the base condition is simply i > 0. The time complexity for Tree Recursion is O(2^n) and it will be the same for all trees as exponential function exceeds every other functions.

Indirect recursion has multiple functions with possibly different base conditions calling each other in cycle. The time complexities of these Indirect Recursions will be the sum of the number of times each function is called, times the time complexity of that function.

Linear Search Algorithm:- sequentially checks for element in array and returns its index if present or returns -1 if absent. The time complexity of linear search is O(n).

Binary Search Algorithm:- works only if the array is already sorted. Checks for middle element and compares with the key and keeps halving the array until the key matches the middle element. The time complexity of binary search is O(logn).

Sort algorithms:-

Stable sorting vs Unstable sorting: If the relative positioning of the duplicate elements within the array is maintained, then the sorting is stable. Otherwise the sorting is unstable. In many cases, the original positioning of the value needs to be preserved as they’re linked to other attributes.

Selection Sort: involves the following:

* Select the minimum element from the collection
* Place the selected element at the appropriate position (first if ascending order) and substitute with that position’s elements
* Repeat with the elements other than this position

It is an unstable sort. Time complexity for comparisons is O(n^2) and for swapping is O(n) for worst case.

Insertion Sort: involves the following:

* The leftmost element of the collection is exported into an empty collection.
* This is repeated for the second and all other elements.
* Post-addition of the second element and so on, the newly added element to the right is compared with its adjacent element and its positions replaced if they’re not in order. This process is done until the element is in order. This chain replacement is done again for the next element addition and so on.

It is a stable sort. Time complexity for comparisons is O(n^2) and for swapping is O(n^2) for worst case. For best case when the collection is already sorted, O(n) is time complexity.

Bubble Sort: involves the following:

* The first and second elements are compared and if they’re in order, the comparison moves to second and third and so on. If they’re not in order, they’re swapped.
* This cycle is repeated until the whole collection is sorted, the last element of the current cycle will be the highest of the collection and can be ignored for subsequent cycles. This will reduce the length of the subsequent cycles.

It is a stable sort. Time complexity for comparisons is O(n^2) and for swapping is O(n^2) for worst case.

Merge Sort: involves the following:

* Divide and conquer approach that uses 2 algorithms: ‘merge-sort’ to break down the collection into subsets (tree shaped connection) and ‘merge’ to combine the subsets back in sorted order.
* The tree shaped connection is regrouped by ‘merge’ algorithm which sorts the individually sorted subsets before joining.
* The ‘merge’ process compares the first elements of both the subsets and selects the lesser element as the joined set’s first element. The leftover element is then compared with the second element of the other subset and so on. The above step is repeated until one of the subsets are depleted. The leftovers of the remaining subset are then added to the joined set.

It is a stable sort. The ‘merge’ algorithm has time complexity of O(n1+n2) or O(n) whereas the ‘merge-sort’ algorithm has the time complexity of O(logn). In tandem, both have a time complexity of O(n\*logn).

Quick Sort: involves the following:

Terminology: A pivot element is the element whose left/previous elements are lesser than itself and the right/next elements are higher than itself (its name is still pivot element as long as the process is ongoing). A sorted position is a pivot element that is satisfies the pivot element definition. A partition position is a pivot element which is in between the collection and not at the start or end.

* Again, this process uses divide and conquer approach. This process/cycle that starts below requires at least two elements be present. If there are no elements or one element, they’re already sorted or in sorted position.
* We begin with the first or last element as the (would-be) pivot element. The rest of the elements are split to before the pivot or after the pivot if they are lesser or greater than the pivot itself.
* There are 2 positions: i and j that are tracked alongside the pivot which is at the first element. The i position starts at the pivot/first position and j starts its position after the last element.
* The i position moves one element at a time towards the end and checks if the element at its position is lesser than the pivot element. If yes, it moves on to the next element. Else no, it stops.
* Once the i position stops, the j position moves one element at a time towards the start and checks if the element at its position is greater than the pivot element. If yes, it moves on the next element. Else if no, it stops (except when it crosses i position, then swap with pivot element).
* After the j position stops, the elements at i and j swap. The process repeats from 2 steps above.
* Continuation of the above 3 steps will have i and j positions cross and once the j position moves past the i position by one element, the cycle stops. The pivot element switches positions with the j position. The pivot element is now in its sorted position and becomes a partition element.
* The collection is split into two across the partition element and the above process is recursively implemented for each collection (not including the partition element).

It is an unstable sort. The best case occurs when the pivot element moves to the center of the collection. Time complexity for the partition function is O(logn) and for the comparisons is O(n), hence the time complexity for the cycles/rounds is O(n\*logn). The worst case occurs when the collection is already sorted either ascending or descending. The time complexity then becomes O(n^2) as it takes O(n) for both comparisons and partitions.

Shell Sort: similar to Insertion Sort and involves the following:

* Compute the gap = length / 2. The divider doesn’t have to be 2, it can be a prime number < total number of elements.
* Consider the first element and the element after the gap from first element, compare them and swap if the first element is higher than the one after gap.
* Next, check the element before the gap from first element and do the comparison and swap if previous element is higher than the first. Since this is impossible for first element, skip this step.
* Using the same gap, continue to the second element and perform the above 2 steps and so on for third and all other elements. This is the end of first cycle.
* For subsequent cycles gap = gap / 2 or gap / prime. It should be consistent with step 1.
* Keep going until gap = 0 and process is completed.

It is an unstable sort. Time complexity for the cycles/rounds is O(n\*logn) for worst case.

Heap Sort: to be learnt with Heaps/Priority Queues DS

Count Sort: is an index-based sorting technique and not elements-based, works only for positive integers and involves the following:

* Create a count array with the limit extending to the highest element of the collection.
* All the elements start as null and are initialized as zero.
* For each occurrence of an element, the corresponding index value of the count array will be incremented. Duplicate elements have the numbers other than one at their index.
* To generate the sorted array, we trace the count array and if the index contains a non-zero element, we add that index to the empty array (sorted array) and decrement the value at that index of count array. After the traversal, the sorted array is complete

This is the fastest sort but will require the most space too. It is also unstable sort. The time complexity of Count Sort is O(n+m) for n as size of array and m as highest element or O(n). Space complexity is also O(n).

Bucket Sort: to be learnt with Hashes DS

Radix Sort: is also an index-based sorting technique and doesn’t compare elements, works only for positive integers and involves the following:-

* Create ten buckets 0-9: one for each digit of the decimal number system.
* Corresponding to the unit’s place of elements in collection, place all of them in their respective buckets (0 is used when digit is unavailable).
* Remove the elements from the buckets in the order 0-9 and reform the collection in the new order.
* Repeat the second and third steps for the ten’s, hundred’s, thousands’s, ten thousand’s so on until the highest element/digit is covered.
* Upon the completion of the last cycle, the reformed collection will be sorted.

The time complexity is O(digits\*n) or O(n). The space complexity is O(n).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithms** | **Best case** | **Average case** | **Worst case** | **Space** | **Stable** |
| Selection Sort | O(n^2) | O(n^2) | O(n^2) | O(1) | No |
| Insertion Sort | O(n) | O(n^2) | O(n^2) | O(1) | Yes |
| Bubble Sort | O(n) | O(n^2) | O(n^2) | O(1) | Yes |
| Shell Sort | O(n\*logn) | O(n\*logn) | O(n^2) | O(1) | No |
| Merge Sort | O(n\*logn) | O(n\*logn) | O(n\*logn) | O(n) | Yes |
| Quick Sort | O(n\*logn) | O(n\*logn) | O(n^2) | O(n) | No |
| Heap Sort | O(n\*logn) | O(n\*logn) | O(n\*logn) | O(1) | No |
| Count Sort | O(n) | O(n) | O(n) | O(n) | Yes |
| Bucket Sort | O(n) | O(n^2) | O(n^2) | O(n) | Yes |
| Radix Sort | O(n) | O(n) | O(n) | O(n) | Yes |