**Algorithmic Toolbox**

Maximum Pair-wise problem:  
We need to find the maximum product of two number from an array of n numbers. The naïve solution would be to use two for loops which is O(n2) time complexity. The product can be greater than the amount an ‘int’ can handle. So use long long. A better solution would be to loop over only once and find the maximum two numbers. This solution would also fail for some inputs. For that, we need to do stress testing.

Stress Testing:  
This is used most of the times on algorithms. Stress testing is a way of testing your algorithm with another alternative algorithm. It can input infinite number of sequences and test all the possibility. The alternative solution which is used should also be provided by the user. Here we can use a slow, brute force solution as the alternative solution to do the stress testing. This will give us the input on which our optimized solution fails.

Reasons for the code to fail:

1. The maximum number occurs twice. For example {2, 9, 3, 1, 9}.

Why study Algorithm?  
For example, we are given a map and we need to find the shortest path between two points. These problems are complicated and simple solutions could be slow and need optimization. This is where algorithms are needed.

Fibonacci Numbers:  
Sequence of natural number where first two numbers are 0 and 1 and rest is the sum of latest two numbers. The naïve algorithm would be to call recursively

If n <= 1:  
 return n;

Else:  
 return F(n-1) + F(n-2);  
This algorithm calculates intermediate values many times. Due to this, the time required is also a sequence of Fibonacci numbers. So we need optimization.   
We can create an array and store n Fibonacci numbers. This can reduce the time.

GCD:  
This is very important in cryptography. We need to use this algorithm on large numbers. A naïve algorithm would be to run from 1 to a+b and find the largest common divisor. An optimized algorithm would be Euclidean Algorithm. This works on the Lemma gcd(a, b) = gcd(b, a%b). This takes log(ab) steps.

Computing Runtimes:  
We cannot figure our runtime for each algorithm. Each computer has their own specifications and their runtimes would be different. So we need to figure out runtimes with respect to the number of inputs. This is called asymptotic notations. This gives us how does runtime scale with input size. Log n < sqrt(n) < n < nlog(n) < n2 < 2n.

Big-O notation:  
f(n) = O(g(n)) if f(n) <= cg(n)

­Greedy Algorithms:  
We can solve many problems like maximum salary problem or queue of patients such that the total wait time is minimized. We can given the treating time of the patients and we have to arrange them in such a way that the total waiting time of all the patients would be less. For this, we can say that we have to arrange the patients with treating time in ascending order so that the total waiting time is minimum. This can be done using two for loops with time – O(n2). But an optimized way is to sort the treating time. This can be done in O(nlogn) time.

Celebration Party Problem:  
Many children came to celebration. We have to organize them into the minimum possible number of groups such that the age of any two children in the same group should differ by atmost two years. A naïve algorithm would be to consider all the groups. So time would be O(2n). Optimized algorithm would be to consider age as points on a line and then make minimum three segments with atmost two points in between.

Algorithm would be:  
Cover the leftmost point with the segment of length 2. Remove all points in this segment. Solve recursively for the rest of the segment.

This is O(n) although it contains two while loops. If we sort as well, then O(nlogn).

Divide and Conquer:  
Not all the problems can be solved using greedy algorithms. So we use divide and conquer. We break the problem down into sub-problems into original problems. Then we combine to get the results. We can solve the sub problems recursively and so we only need to set the base condition, rest is called by recursive functions.

Linear Search: We search for an element by going from left to right. This can be time consuming if there are very large arrays.

Binary Search: Here, we compare the element to find with the middle element. If the element to search is known to occur before middle element, then we break the problem and search the element in the first half array. The condition for this is that the array should be sorted.

Polynomial Multiplication: Here we can store coefficients of the polynomial in an array. Then we can use two loops to calculate the coefficient of the resultant polynomial. This runs in O(n2) time. We can use divide and conquer to solve faster.

Master Theorem for Recurence:  
T(n) = aT(ceil(n/b)) + O(nd) for a >= 0, b > 1 and d >= 0. Then

T(n) = O(nd) if d > logb(a)  
T(n) = O(nd logn) if d = logb(a)  
T(n) = O( if d < logb(a)

Sorting:  
Sorting is an important step of many efficient algorithm. Sorted data allows for more efficient queries. For example *Binary Search*.

Selection Sort: We find the minimum element, swap it with the first element. Then do the same with the rest array (i.e. array without the first element). An important property of this algorithm is that the running time of the algorithm does not depend if array is already sorted, or in reverse order, or just random. Time – O(n2). Sorts the array *in place*.

Merge Sort: Based on divide and conquer. We split the array in two equal parts. We sort the two half array by recursive call. Then we merge the two sorted array with a new function sortedMerge. To merge the two array, time required is O(n). But for sorting the two half arrays we need time O(nlog n). This is asymptotically optimum. No sorting algorithm can do better than this.

Non-Comparison Based Sorting Algorithm: Here, if we have small integers, then we can solve the array in sort time O(n). But, here we know about array. We travel the array once and store the number of occurance of each of the elements. Then, we create an array with that information. This is called Counting Sort.

Quick Sort: This is the most used algorithm. Time – O(nlog n) (on average). Here, we start with the first element. We then convert the array such that all the elements smaller than the first element are before the first element and all element greater are right of the first element. Then we recursively call for the array on the left and on the right. The running time of the algorithm depends on the pivot. But on an average it takes O(nlog n). The worst time is O(n2). The average case is based on the assumption that the pairwise elements are different. If we have multiple occurance, then we partition into three parts. First, less than, third, greater than and middle equal element.

Dynamic Programming:  
We saw that the greedy algorithm of coin denomination failed. It also cannot be solved with recursion. So we have to perform dynamic approach. We solve the recursion problem and store their answer which can be used again and again hence reducing the number of computation.

All of the above notes were till mid-term submission.

Here is the code which I wrote for the programming assignments.

Link: [Data Structure and Algorithms](https://iitbacin-my.sharepoint.com/:f:/g/personal/20d180023_iitb_ac_in/EvOe206hCw9Oqg4Dyeb6R_ABiA6U8RyJWlwoRaVB0Hp4dg?e=lifrzY)