Algorithm classification

- Algorithms that use a similar problem-solving approach can be grouped together

- The purpose is not to be able to classify an algorithm as one type or another, but to highlight the various ways using which a problem can be solved

- Algorithm types we will consider include:
  - Brute force algorithms
  - Greedy algorithms
  - Simple recursive algorithms
  - Backtracking algorithms
  - Divide and conquer algorithms
  - Dynamic programming algorithms
  - Randomized algorithms

Brute force algorithm

- Brute Force is a straightforward approach of solving a problem based on the problem statement. It is one of the easiest approaches to solve a particular problem.

- It is useful for solving small size dataset problem.

- Most of the times, other algorithm techniques can be used to get a better solution of the same problem

Brute force algorithm

- Brute force is the first algorithm that comes into mind when we see some problem.
- They are the simplest algorithms that are very easy to understand.
- These algorithms rarely provide an optimum solution. Many cases we need to find other effective algorithm that is more efficient than the brute force method.

Greedy algorithms

- Greedy algorithms are generally used to solve optimization problems.
- To find the solution that minimizes or maximizes some value (cost/profit/count etc.).
- In greedy algorithm, solution is constructed through a sequence of steps.
  - At each step, choice is made which is locally optimal.
  - We always take the next data to be processed depending upon the dataset which we have already processed and then choose the next optimum data to be processed.
- Greedy algorithms may not always give optimum solution.
- The main advantage of the Greedy method is that it is straightforward, easy to understand and easy to code.

Greedy algorithms

- Some examples of Greedy algorithms are:
    - Minimal spanning tree: Prim's algorithm, Kruskal's algorithm
    - Dijkstra's algorithm for single-source shortest path

Simple recursive algorithms

- A simple recursive algorithm:
    - Solves the base cases directly
    - Recurs with a simpler subproblem
    - Does some extra work to convert the problem to the simpler subproblem

Example recursive algorithms

- To count the number of elements in a list:
  - If the list is empty, return zero; otherwise,
  - Step past the first element, and count the remaining elements in the list
  - Add one to the result
- To test if a value occurs in a list:
  - If the list is empty, return false; otherwise,
  - If the first thing in the list is the given value, return true; otherwise
  - Step past the first element, and test whether the value occurs in the remainder of the list

Backtracking algorithms

- Backtracking algorithms are based on a depth-first recursive search
- A backtracking algorithm:
    - Tests to see if a solution has been found, and if so, returns it; otherwise
    - For each choice that can be made at this point,
        - Make that choice
        - Recur
        - If the recursion returns a solution, return it
    - If no choices remain, return failure
- Example: searching key of a lock from available bunch of keys.

Divide and Conquer

- Divide-and-Conquer algorithms works by recursively breaking down a problem into two or more subproblems (divide), until these sub problems become simple enough so that can be solved directly (conquer).
- The solution of these sub problems is then combined to give a solution of the original problem.
- Divide-and-Conquer algorithms involve basic three steps
  - Divide the problem into smaller problems.
  - Conquer by solving these problems.
  - Combine these results together.
- In divide-and-conquer the size of the problem is reduced by a factor (half, one-third etc.), While in decrease-and-conquer the size of the problem is reduced by a constant.

Divide and Conquer

- Examples of divide-and-conquer algorithms:
  - Merge-Sort algorithm (recursion)
  - Quicksort algorithm (recursion)
  - Computing the length of the longest path in a binary tree (recursion)
  - Computing Fibonacci numbers (recursion)

- Examples of decrease-and-conquer algorithms:
  - Computing POW(a, n) by calculating POW(a, n/2) using recursion
  - Binary search in a sorted array (recursion)
  - Searching in BST

Dynamic programming algorithms

- A dynamic programming algorithm remembers past results and uses them to find new results
- Dynamic programming is generally used for optimization problems
  - Multiple solutions exist, need to find the “best” one
  - Requires “optimal substructure” and “overlapping subproblems”
    - Optimal substructure: Optimal solution contains optimal solutions to subproblems
    - Overlapping subproblems: Solutions to subproblems can be stored and reused in a bottom-up fashion
- . Dynamic Programming (DP) is a simple technique but it can be difficult to master.

Dynamic programming algorithms

- Dynamic programming and memoization work together.
- By using memorization [maintaining a table of sub problems already solved], dynamic programming reduces the exponential complexity to polynomial complexity ($O(n^2)$, $O(n^3)$, etc.) for many problems.
- The major components of DP are:
  - Recursion: Solves sub problems recursively.
  - Memorization: Stores already computed values in table (Memoization means caching).

Dynamic Programming = Recursion + memoization

Let us take an example

Find Maximum Value Contiguous Subsequence: Given an array of n numbers, give an algorithm for finding a contiguous subsequence A(i)... A(j) for which the sum of elements is maximum.

Example: {-2, **11, -4, 13**, -5, 2} → 20 and {1, -3, **4, -2, -1, 6**} → 7

Note: The algorithms doesn't work if the input contains all negative numbers. It returns 0 if all numbers are negative.

Example: {-2, **11, -4, 13**, -5, 2} → 20 and {1, -3, **4, -2, -1, 6**} → 7

```
int MaxContigousSum(int A[], in n) {
    int maxSum = 0;
    for(int i = 0; i < n; i++)                          // for each possible start point
        for(int j = i; j < n; j++)    {                 // for each possible end point
            int currentSum = 0;
            for(int k = i; k <= j; k++)
                currentSum += A[k];
            if(currentSum > maxSum)
                maxSum = currentSum;
        }
    }
    return maxSum;
}
```

Time Complexity: $O(n^3)$. Space Complexity: $O(1)$.

Example: {-2, **11, -4, 13**, -5, 2} → 20 and {1, -3, **4, -2, -1, 6**} → 7

```
int MaxContigousSum(int A[], int n) {
    int maxSum = 0;
    for( int i = 0; i < n;   i++) {
        int currentSum = 0;
        for( int j = i; j < n; j++)    {
            currentSum += a[j];
            if(currentSum > maxSum)
                maxSum = currentSum;
        }
    }
    return maxSum;
}
```

Time Complexity: $O(n^2)$. Space Complexity: O(1).

Example: {-2, **11, -4, 13**, -5, 2} → 20 and {1, -3, **4, -2, -1, 6**} → 7

```
int MaxContigousSum(int A[], int n) {
    int M[n] = 0, maxSum = 0;
    if(A[0] > 0)
            M[0] = A[0];
    else M[0] = 0;
    for( int i = 1; i < n; i++) {
            if( M[i-1] + A[i] > 0)
                    M[i] = M[i-1] + A[i];
            else    M[i] = 0;
    }
    for( int i = 0; i < n; i++)
            if(M[i] > maxSum)
                    maxSum = M[i];
    return maxSum;
}
```

Time Complexity: O(n). Space Complexity: O(n).

Example: {-2, **11, -4, 13**, -5, 2} → 20 and {1, -3, **4, -2, -1, 6**} → 7

```
int MaxContigousSum(int A[], int n) {
    int sumSoFar = 0, sumEndingHere = 0;
    for(int i = 0; i < n; i++)   {
            sumEndingHere = sumEndingHere + A[i];

            if(sumEndingHere < 0) {
                    sumEndingHere = 0;
                    continue;
            }
            if(sumSoFar < sumEndingHere)
                    sumSoFar = sumEndingHere;
    }
    return sumSoFar;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Examples of Dynamic Programming Algorithms:

- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, edit distance.

- Algorithms on graphs can be solved efficiently: Bellman-Ford algorithm for finding the shortest distance in a graph

- Subset Sum

Stochastic and randomized algorithms

- A randomized algorithm uses a random number at least once during the computation to make a decision
  - Example: In Quicksort, using a random number to choose a pivot
  - Example: Trying to factor a large prime by choosing random numbers as possible divisors

Thank You!!!!!