Algorithm By Astik Anand (Microsoft SDE2)

Contents:

1. Complexity Analysis
2. Searching Algorithm
3. Sorting Algorithm
4. Selection Algorithm
5. Recursion Backtrack Approach
6. Greedy Approach
7. Dynamic Programming Approach
8. Divide and Conquer Approach
9. Pattern Search Algorithms
10. Bit Algorithm
11. Mathematical Algorithms
12. Geometrical Algorithms
13. Randomized Algorithm
14. Branch and Bound Approach
15. Miscellaneous Algorithm Problems

Complexity Analysis

Why Performance Analysis?

- There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc.
- Why to worry about performance?
    - The answer to this is simple, we can have all the above things only if we have performance.
    - So performance is like currency through which we can buy all the above things.
    - Another reason for studying performance is – speed is fun!
- To summarize, performance == scale. Imagine a text editor that can load 1000 pages, but can spell check 1 page per minute OR an image editor that takes 1 hour to rotate your image 90 degrees left OR … you get it.
- If a software feature can not cope with the scale of tasks users need to perform – it is as good as dead.

2 algorithms for a task, how to find which one is better?

- **Naive Approach:**
    - Implement both the algorithms and run the two programs on the computer for different inputs and see which one takes less time.
    - **Problems with this approach:**
        - It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
        - It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

- **Asymptotic Analysis:**
    - Discussed in detail below.

1. Asymptotic Analysis

What is this Asymptotic Analysis?

- It is the big idea that handles above issues in analyzing algorithms.
- Here we evaluate the performance of an algorithm **in terms of input size** (we don't measure the actual running time).

- We calculate, how does the time (or space) taken by an algorithm increases with the input size.
- **Example:** Let us consider the search problem (searching a given item) in a sorted array.
    - One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic).
    - To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms:
        - Let us say we run the Linear Search on a fast computer and Binary Search on a slow computer.
        - For small values of input array size n, the fast computer may take less time.But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine.
        - The reason is the order of growth of Binary Search with respect to input size logarithmic while the order of growth of Linear Search is linear.
        - So the machine dependent constants can always be ignored after certain values of input size.

Does Asymptotic Analysis always work?

- Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms.
- **Example:** say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine.
    - Both of these algorithms are asymptotically same (order of growth is nLogn).
    - So, with Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis.
- Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value.
- It might be possible that those large inputs are never given to our software and an algorithm which is asymptotically slower, always performs better for that particular situation.
- So, we may end up choosing an algorithm that is Asymptotically slower but faster for our software.

Worst, Average and Best Case

- **Worst Case Analysis (Usually Done)**
- **Average Case Analysis (Sometimes done)**
- **Best Case Analysis (Bogus)**

Asymptotic Notations

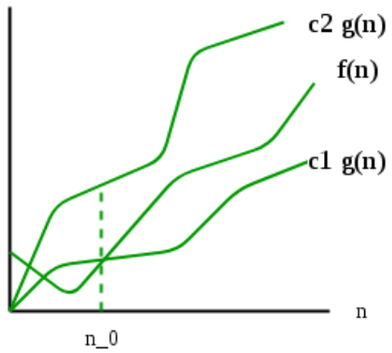
Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.

- **1. Θ Notation:**

  The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.

  For a given function g(n), we denote Θ(g(n)) is following set of functions.

  Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 <= c1*g(n)* <= *f(n)* <= *c2*g(n) for all n >= n0}

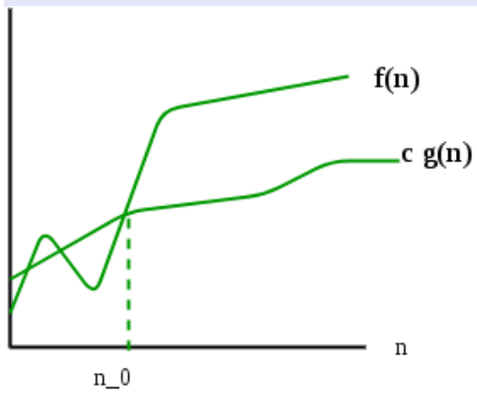


f(n) = theta(g(n))

- **2. Big O Notation:**

  The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.

  O(g(n)) = { f(n): there exist positive constants c and n0 such that 0 <= f(n) <= c*g(n) for all n >= n0}



f(n) = Omega(g(n))

- **3. Ω Notation**

  Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

  Ω (g(n)) = {f(n): there exist positive constants c and n0 such that 0 <= c*g(n) <= f(n) for all n >= n0}.

Calculating Time Complexity

- **1. O(1):**

  Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

```
// set of non-recursive and non-loop statements
```

Example: swap() function has O(1) time complexity.

- **2. O(n):**

  Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount.

  Example: following functions have O(n) time complexity.

```
// Here c is a positive integer constant

for (int i = 1; i <= n; i += c) {
  // some O(1) expressions
}
```

- **3. O(n^c):**

  Time complexity of nested loops is equal to the number of times the innermost statement is executed.

  Example: The following sample loops have O(n^2) time complexity

```
for (int i = 1; i <=n; i += c) {
  for (int j = 1; j <=n; j += c) {
    // some O(1) expressions
  }
}
```

- **4. O(logn):**

  Time Complexity of a loop is considered as O(Logn) if the **loop variables is divided / multiplied by a constant amount**.

```
for (int i = 1; i <=n; i *= c) {
  // some O(1) expressions
}

for (int i = n; i > 0; i /= c) {
  // some O(1) expressions
}
```

  Example: Binary Search(refer iterative implementation) has O(Log n) time complexity.

  Let us see mathematically how it is O(Log n): The series that we get in first loop is 1, c, c2, c3, … ck. So now, ck = n then k = logcn and hence O(Log n)

- **5. O(loglogn):**

  Time Complexity of a loop is considered as O(LogLogn) if the **loop variables is reduced / increased exponentially by a constant amount**.

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {
  // some O(1) expressions
}

//Here root is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = root(i)) {
```

```
    // some O(1) expressions
}
```

- **Solving Recurrences**

  Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity.

  Example: Merge Sort: **T(n) = 2T(n/2) + cn**

  There are many other algorithms like Binary Search, Tower of Hanoi, etc.

  Methods to solve Recurrences:

  - **1. Substitution Method (Method of Guessing and Confirm):**

    We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

    **Example: ** Consider the recurrence T(n) = 2T(n/2) + n

    We guess the solution as T(n) = O(nLogn). Now we use induction to prove our guess.

    We need to prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.

    T(n) = 2T(n/2) + n

    <= cn/2Log(n/2) + n

    = cnLogn - cnLog2 + n

    = cnLogn - cn + n

    <= cnLogn

  - **2. Master Theorem Method**

Master Theorem

Divide & Conquer

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$$

$a \geq 1 \quad b > 1, \quad k \geq 0 \quad p \in R$

1) If $a > b^k$: $T(n) = \Theta(n^{\log_b a})$
2) If $a = b^k$:
   a) If $p > -1$: $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) If $p = -1$: $T(n) = \Theta(n^{\log_b a} \log \log n)$
   c) If $p < -1$: $T(n) = \Theta(n^{\log_b a})$
3) If $a < b^k$:
   a) If $p \geq 0$: $T(n) = \Theta(n^k \log^p n)$
   b) If $p < 0$: $T(n) = \Theta(n^k)$

Subtract & Conquer

$$T(n) = \begin{cases} c & \text{if } n \leq 1 \\ aT(n-b) + f(n) & \text{if } n > 1 \end{cases}$$

$c, a > 0 \quad b > 0 \quad k \geq 0 \quad$ & $f(n) = O(n^k)$

$$T(n) = \begin{cases} O(n^k) & \text{if } a < 1 \\ O(n^{k+1}) & \text{if } a = 1 \\ O(n^k a^{\frac{n}{b}}) & \text{if } a > 1 \end{cases}$$

Subtract & Conquer Variant

$T(n) = T(\alpha n) + T((1-\alpha)n) + \beta n$

$0 < \alpha < 1 \quad \beta > 0$

$T(n) = O(n \log n)$

Master Theorem

$$T(n) = aT(n/b) + \Theta(n^c)$$

$a > 1, b > 1$

1) $c < \log_b a$: $\Theta(n^{\log_b a})$
2) $c = \log_b a$: $\Theta(n^c \log n)$
3) $c > \log_b a$: $\Theta(n^c)$

$$T(n) = aT(n-b) + O(n^c)$$

$a > 0, b > 0, c \geq 0$

1) $a < 1$: $O(n^c)$
2) $a = 1$: $O(n^{c+1})$
3) $a > 1$: $O(n^c a^{n/b})$

2. Amortized Analysis

What is this Amortized Analysis?

- It refers to determining the time-averaged running time for a sequence of operations.
- It is worst-case analysis for a sequence of operations rather than for an individual operation.
- **Example:** Finding k^{th} smallest element
  - We can solve this by sorting the array and after sorting we just need to return kth element.
  - Sorting takes O(nlogn) time so for individual operation or Asymptotic analysis time is O(nlogn).
  - But if we need to take the same operation for n times the amortized time-complexity = O(nlogn/n) = O(logn).
  - Hence, sorting once has reduced the complexity of subsequent operations.
- The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

Why Amortized Analysis?

- Motivation for amortized analysis is to better understand the running time of certain techniques, where standard worst case analysis provides an overly pessimistic bond.
- Applies to method that consists of sequence of operations, where the vast majority of the operations are cheap, but some of the operations are expensive.
- If we can show that the expensive operations are particularly rare we can charge them to the cheap operations, and only bound the cheap operations.

How to calculate it?

- General approach is to assign an **artificial cost (amortized cost)** to each operation in the sequence of operations.
- Such that the total of the artificial costs for sequence of operations bounds total of real costs for the sequence.
- Amortized analysis thus is a correct way of understanding the overall running time.

Benefits:

- When one event in a sequence affects the cost of later events:
  - One particular task may be expensive.
  - But it may leave data structure in a state that next few operations become easier.

3. Space Complexity

What is Space Complexity?

- The term Space Complexity is misused for Auxiliary Space at many places.
- **Auxiliary Space:** is the extra space or temporary space used by an algorithm.
- **Space Complexity:** of an algorithm is total space taken by the algorithm with respect to the input size and includes both Auxiliary space and space used by input.
- But if we want to compare algorithms on the basis of space, then Auxiliary Space would be a better criteria than Space Complexity.
    - **Example:** Merge Sort uses O(n) auxiliary space, Insertion sort and Heap Sort use O(1) auxiliary space. Space complexity of all these sorting algorithms is O(n) though.

4. Complexity Classes

What are complexity classes?

- In computer science, in order to understand the problems for which solutions are not there, the problems are divided into classes k/a Complexity Classes.
- In complexity theory, a complexity class is a set of problems with related complexity.
- It is the branch of theory of computation that studies the resource required during computation to solve a given problem.
- The most common resources are time (how much time the algorithm takes to solve a problem) and space (how much memory it takes).

Easy Problems & Hard Problems

- The classification is done on based on the running time (or memory) that an algorithm takes for solving the problem.
- Problems with lower rate of growth are called **easy problems** (easy solved problems).
- Problems with higher rate of growth are called **hard problems** (hard solved problems).

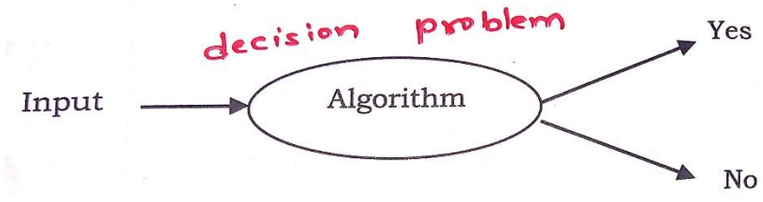
Time Complexity	Name	Example	Problems
O(1)	Constant	Adding an element to the front of a linked list	Easy solved problems
O($logn$)	Logarithmic	Finding an element in a binary search tree	
O(n)	Linear	Finding an element in an unsorted array	
O($nlogn$)	Linear Logarithmic	Merge sort	
O(n^2)	Quadratic	Shortest path between two nodes in a graph	
O(n^3)	Cubic	Matrix Multiplication	
O(2^n)	Exponential	The Towers of Hanoi problem	Hard solved problems
O($n!$)	Factorial	Permutations of a string	

Polynomial & Exponential Time

- **Exponential time** means, in essence, trying every possibility (Example: Backtracking Algorithms) and they are very slow in nature.
- **Polynomial time** means having some clever algorithm to solve a problem, and we don't try every possibility.
    - **Polynomial time:** O(nk) for some k.
    - **Exponential time:** O(kn) for some k.

Decision Problem & Decision Procedure

- A **decision problem** is a question with yes/no answer and the answer depends on the values of the input.
    - **Example:** "Given an array of n numbers check whether there are any duplicates or not?" is a decision problem and the answer can be "Yes/No" depending on input.
- Solving a given decision problem with an algorithm is called **decision procedure** for that problem.



Types of Complexity Classes

P Class

- Set of decision problems that can be solved by a **deterministic machine in polynomial time** (P stands for polynomial time).
- These are set of problems whose solutions are easy to find.

NP Class

- Set of decision problems that can be solved by a **non-deterministic machine in polynomial time** (NP stands for non-deterministic polynomial time)
- These are set of problems whose solutions are hard to find but easy to verify.
- If someone gives us a solution to the problem, we can tell them whether it is right or not in polynomial time.
- If the answer to a problem is "YES", then there is a proof of this fact, which can be verified in polynomial time.

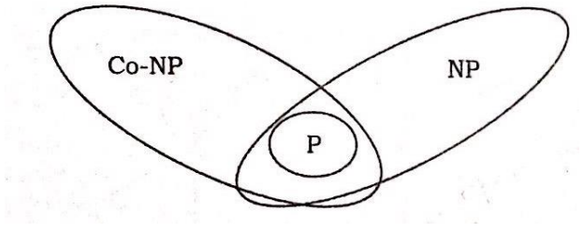
Co-NP Class

- Opposite or complement of NP.
- If the answer to a problem Co-NP is "NO", then there is a proof of this fact that can be checked in polynomial time.

P	Solvable in polynomial time
NP	"YES" answers can be checked in polynomial time.
Co-NP	"NO" answers can be checked in polynomial time.

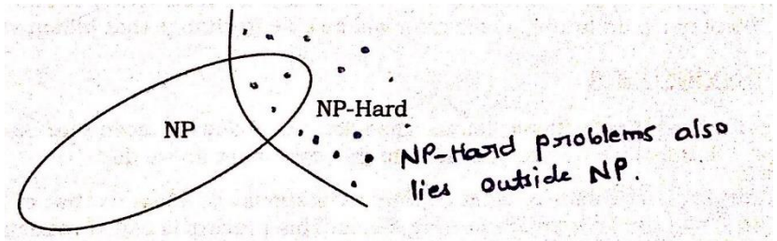
Relationship between P, NP, Co-NP:

- Any problem in P is also in NP coz if a problem is in P, we can verify "YES" answer in polynomial time.
- Similarly any problem in P is also in Co-NP coz if a problem is in P, we can verify "NO" answer in polynomial time.
- **One of the important open questions in theoretical computer science is whether or not P=NP (Nobody knows).**
- Intuitively it should be obvious that **P≠NP**, but nobody knows how to prove it.
- **Another open questions is whether NP and Co-NP are different (Nobody knows).**
- Even if we can verify every "YES" answer quickly, there's no reason to think we can also verify "NO" answers quickly.
- It is generally believed that NP≠Co-NP, but again nobody knows how to prove it.



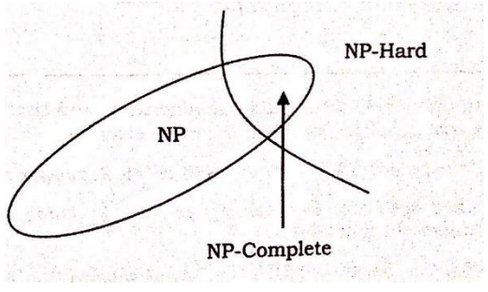
NP-Hard Class

- Class of decision problems which are at least as hard as the hardest problems in NP.
- Every Problem in NP can be reduced to it.
- A problem K is NP-Hard indicates that if a polynomial-time algorithm exists for K then a polynomial-time algorithm exists for every problem in NP.
- K is NP-hard implies that if K can be solved in polynomial-time, then P=NP.
- Although it is suspected that there are no polynomial-time algorithms for NP-hard problems, this has not been proven.
- All NP-Hard problems are not in NP, so it takes a long time to even check them (forget about solving) and it may not even be decidable.
- **NP-hard are not only restricted to decision problems, for instance it also includes search problems, or optimization problems.**
    - Subset-Sum (Decision Problem),
    - Travelling Salesman (Optimization Problem)
    - Halting Problem (Undecidable, Not NP-Complete)



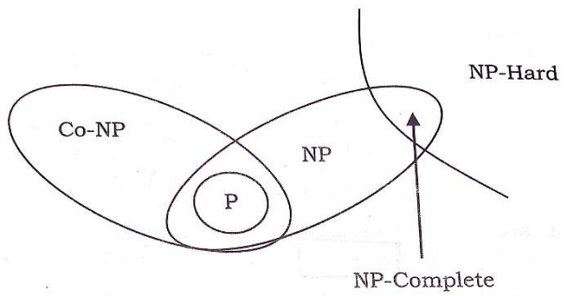
NP-Complete Class

- A problem is NP-Complete if it is part of both NP-hard and NP.
- Class of decision problems which contains the hardest problems in NP (but remember there can be even more harder problem outside NP in NP-Hard class).
- Each NP-complete problem has to be in NP.
- If anyone finds a polynomial-time algorithm for one NP-Complete problem, then we can find polynomial-time algorithm for every NP-Complete problem.
- We can check an answer fast and every problem in NP reduces to it.
- **Example:** Subset-Sum (NP-Complete)



Relationship between P, NP, Co-NP, NP-Hard and NP-Complete

- NP-Complete problems are strict subset of problems that are NP-Hard or NP-Hard is strict superset of NP-Complete.
- Some problems are NP-Hard but not in NP.
- NP-Hard problems might be impossible to solve in general.
- We can tell the difference in difficulty b/w NP-Hard and NP-Complete coz NP includes everything easier than it's toughest problem.
- But if a problem is not in NP, it is harder than all problems in NP.



Does P=NP?

- If P=NP, it means that every problem that can be checked quickly can be solved quickly (remember the difference b/w checking and actually solving).
- This is a big question but nobody knows the answer, coz right now there are lots of NP-Complete problems that can't be solved quickly.
- P=NP means there is a way to solve them fast, quickly means no trial-and-error.
- It could take billion years, but as long as we didn't use trail and error it was quick, coz in future a fast computer will be able to change that billion years to few minutes.

Pseudo-Polynomial Algorithms

What is Pseudo-polynomial?

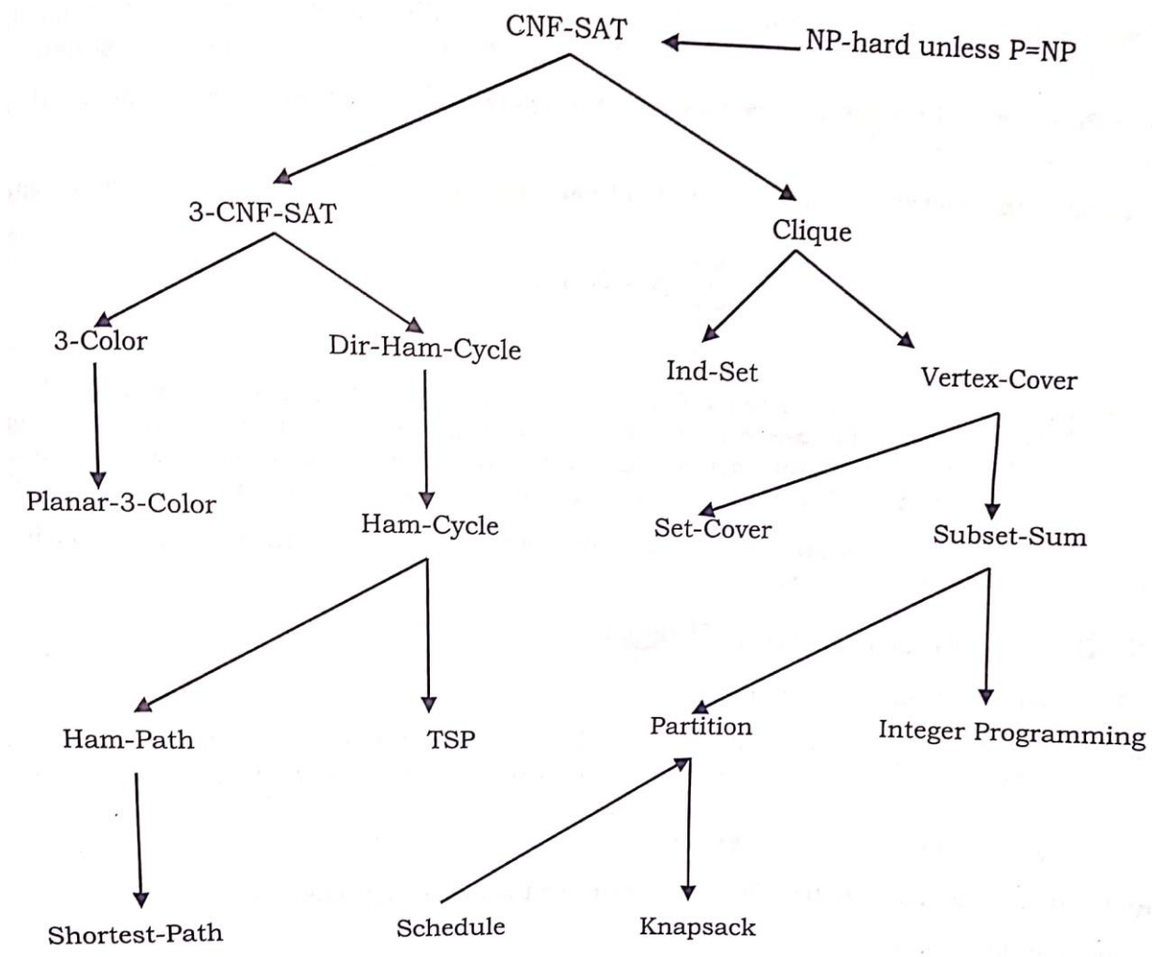
- An algorithm whose worst case time complexity depends on numeric value of input (not number of inputs) is called Pseudo-polynomial algorithm.
- **Example:** Consider the problem of counting frequencies of all elements in an array of positive numbers.
  - A pseudo-polynomial time solution for this is to first find the maximum value, then iterate from 1 to maximum value and for each value, find its frequency in array.
  - This solution requires time according to maximum value in input array, therefore pseudo-polynomial.
  - On the other hand, an algorithm whose time complexity is only based on number of elements in array (not value) is considered as polynomial time algorithm.

Pseudo-polynomial and NP-Completeness

- Some NP-Complete problems have Pseudo Polynomial time solutions.

- **Example:** Dynamic Programming Solutions of **0-1 Knapsack**, **Subset-Sum** and **Partition problems** are Pseudo-Polynomial.
- NP complete problems that can be solved using a pseudo-polynomial time algorithms are called **weakly NP-complete**.

NP-Hard Class Problems



Complexity Analysis Problems

Problem-1:

```
void fun(){
   int i, j;
   for (i=1; i<=n; i++)
      for (j=1; j<=log(i); j++)
         printf("GeeksforGeeks");
}
/* Θ(log 1) + Θ(log 2) + Θ(log 3) + . . . . + Θ(log n) = Θ(log n!) = Θ(nlogn)   coz n!≈nn/2
T(n) = Θ(nlogn) */
```

Problem-2:

```
void fun(int n){
   int j = 1, i = 0;
   while (i < n){
       // Some O(1) task
       i = i + j;
       j++;
   }
}

/* The value of i is incremented by 1,2,3,4, . . . and so on
So the value of i will be like 0, 1, 3, 6, 10, 15, . . .  : We can see that kth term will be k(k+1)/2
k(k+1)/2 = n => k2/2 = n => k = √n
T(n) = Θ(√n) */
```

Searching Algorithms

Why Searching Algorithms ?

- Many of the computer science problems require us to search through the data to find certian things or pattern.
- In these scenarios we need to use the Searching Algorithms.

Famous Searching Algorithms:

- Linear Search
- Binary Search
- Ternary Search

- Jump Search

Algo-1: Binary Search***

Given a sorted array arr[ ] of n elements, write a function to search a given element x in arr[ ].

Return the index if it is found or else return -1.

Algorithm: Divide and Conquer Approach

- Check the middle element of the array, if **middle element = num** return index.
- If the **middle element < num**, search the num in right half of the array.
- Else search in the num in left half of the array.
- If num not found return -1.

Implementation:

```
def binary_search(arr, l, r, num):
    if(r>=l):
        mid = (l+r)//2
        # If mid element is equal to num, return index
        if(arr[mid]==num):
            return mid
        # If mid element is lesser than num, search in the right half of the array
        elif(arr[mid] < num):
            return binary_search(arr, mid+1, r, num)
        # If mid element is grater searhc in left half of the array
        else:
            return binary_search(arr, l, mid-1, num)

    return -1


print("Example-1: binary_search(arr, num)")
arr = [2, 3, 4, 10, 40]
print(binary_search(arr, 0, 4, 10))

print("Example-2: binary_search(arr, num)")
arr = [2, 3, 4, 10, 40]
print(binary_search(arr, 0, 4, 15))

print("Example-3: binary_search(arr, num)")
arr = [2]
print(binary_search(arr, 0, 0, 2))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 13:47:53 ~/Personal/Notebooks/Algorithms/2. Searching & Sorting $ python3 1_binary_search.py
Example-1: binary_search(arr, num)
3
Example-2: binary_search(arr, num)
-1
Example-3: binary_search(arr, num)
0
```

Complexity:

- **Time: O(logn)**
- **Auxilliary Space: O(1)**

Standard Searching Algorithms Problems

1. Search in sorted and rotated array***

Problem:

An element in a sorted array can be found in O(log n) time via binary search. But suppose we rotate an ascending order sorted array at some pivot unknown to you beforehand. So for instance, 1 2 3 4 5 might become 3 4 5 1 2. Devise a way to find an element in the rotated array in O(log n) time.

Example:

Input: arr[] = {5, 6, 7, 8, 9, 10, 1, 2, 3} key = 3 ***Output:*** Found at index 8

Input: arr[] = {5, 6, 7, 8, 9, 10, 1, 2, 3} key = 4 ***Output:*** Not found

Input: arr[] = {30, 40, 50, 10, 20} key = 10 ***Output:*** Found at index 3

Approach-Distorted Binary Search

- Find middle point mid = (l + h)/2
- If key is present at middle point, return mid.
- Else If arr[l..mid] is sorted
    - a) If key to be searched lies in range from arr[l] to arr[mid], recur for arr[l..mid].

    - b) Else recur for arr[mid+1..h]
- Else (arr[mid+1..h] must be sorted)
    - a) If key to be searched lies in range from arr[mid+1] to arr[h], recur for arr[mid+1..h].
    - b) Else recur for arr[l..mid]

Implementation

```
def search(arr, start, end, key):
    if start > end:
        print("Key {} : NOT Found".format(key))
        return

    mid = (start + end) // 2
    if arr[mid] == key:
        print("Key {} : Found at index {}".format(key, mid))
        return

    # If arr[start...mid] i.e 1st half is sorted
    if arr[start] <= arr[mid]:
        # As the 1st subarray is sorted, Quickly check if key lies in first half or 2nd half
        if key >= arr[start] and key <= arr[mid]:
            return search(arr, start, mid-1, key)
        else:
            return search(arr, mid+1, end, key)
    # Else arr[start..mid] is not sorted, then arr[mid... end] must be sorted
    else:
        # As the 2nd subarray is sorted, Quickly check if key lies in 2nd half or first half
        if key >= arr[mid] and key <= arr[end]:
            return search(arr, mid+1, end, key)
        else:
            return search(arr, start, mid-1, key)


print("Example-1: search([5, 6, 7, 8, 9, 10, 1, 2, 3], 0, 8, 3)")
search([5, 6, 7, 8, 9, 10, 1, 2, 3], 0, 8, 3)

print("\nExample-2: search([5, 6, 7, 8, 9, 10, 1, 2, 3], 0, 8, 4)")
search([5, 6, 7, 8, 9, 10, 1, 2, 3], 0, 8, 4)

print("\nExample-3: search([30, 40, 50, 10, 20], 0, 4, 10)")
search([30, 40, 50, 10, 20], 0, 4, 10)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 11:59:11 ~/Personal/Notebooks/Data Structures/1. Array $ python3 2_search_in_sorted_rotated_array.py
Example-1: search([5, 6, 7, 8, 9, 10, 1, 2, 3], 0, 8, 3)
Key 3 : Found at index 8

Example-2: search([5, 6, 7, 8, 9, 10, 1, 2, 3], 0, 8, 4)
Key 4 : NOT Found

Example-3: search([30, 40, 50, 10, 20], 0, 4, 10)
Key 10 : Found at index 3
```

Complexity:

- **Time: O(LogN)**
- **Auxilliary Space: O(1)**

Sorting Algorithms

Why Sorting Algorithms ?

Many a times a computer science problems need data to be present in certain sorted order. These are the scenarios when we need the Sorting Algorithms.

Famous Sorting Algorithms:

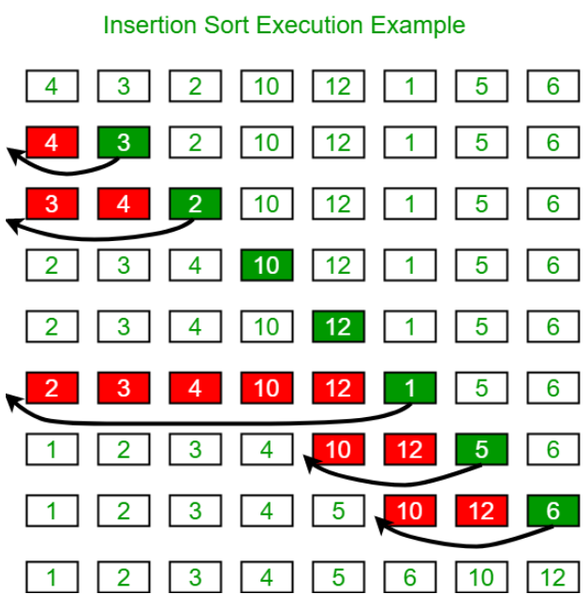
- **Comparison Sorts:**
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - Heap Sort
  - Binary Tree Sort

- **Non-comparison Sorts:**
  - Bucket Sort
  - Counting Sort
  - Radix Sort
  - Pigeonhole Sort

Algo-1: Insertion Sort

What is Insertion Sort ?

- Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.



Algorithm:

- Loop from i = 1 to n-1.
    - Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

Implementation:

```
def insertion_sort(arr):
    n = len(arr)

    for i in range(n):
        key = arr[i]

        # Now find a place to insert this key by shifting every left element which is larger to right by 1
        # Here we are using linear search to find a place to insert key in sorted arr till i-1
        # We can also use binary search to find that particular place :--> Binary Insertion Sort
        j = i-1
```

```
        while(j>=0 and arr[j]> key):
            arr[j+1] = arr[j]
            j -= 1

        # arr[j] is smaller than or equal to key, so insert key after it
        arr[j+1] = key

    return arr



print("Example-1: insertion_sort(arr)")
arr = [4, 3, 2, 10, 12, 1, 5, 6]
print(insertion_sort(arr))

print("Example-2: insertion_sort(arr)")
arr = [12, 11, 13, 5, 6]
print(insertion_sort(arr))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 14:20:07 ~/Personal/Notebooks/Algorithms/2. Searching & Sorting $ python3 2_insertion_sort.py
Example-1: insertion_sort(arr)
[1, 2, 3, 4, 5, 6, 10, 12]
Example-2: insertion_sort(arr)
[5, 6, 11, 12, 13]
```

Complexity:

- **Time:-** Worst Case:**$O(n^2)$** Average Case: **$O(n^2)$**
- **Auxilliary Space: O(1)**

Notes:

- **Algorithmic Paradigm:** Incremental Approach
- **Sorting In Place:** Yes
- **Stable:** Yes

Uses:

- Insertion sort is used when number of elements is small.
- It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

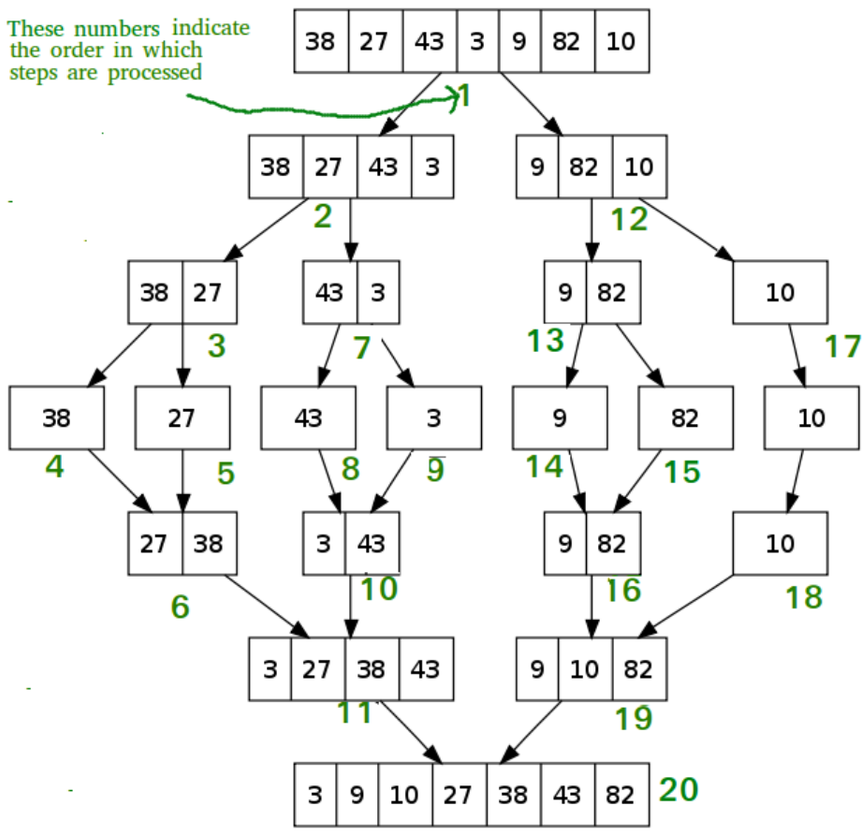
Binary Insertion Sort:

- We can use binary search to reduce the number of comparisons in normal insertion sort.
- Binary Insertion Sort uses binary search to find the proper location to insert the selected item at each iteration.
- Worst Case:**O(nlogn)** Average Case: **O(n^2)**

Algo-2: Merge Sort***

What is Merge Sort ?

- Merge Sort is a Divide and Conquer algorithm.
- It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.
- The **merge( )** function is used for merging two halves.
- The merge( ) is key process that assumes that arr[…mid] and arr[mid+1….] are sorted and merges the two sorted sub-arrays into one.



Algorithm:

- Divide the array into two halves:
- sorted_arr1 = merge_sort for first half
- sorted_arr2 = merge_sort for second half
- sorted_arr = merge(sorted_arr1, sorted_arr2)

Implementation:

```
def merge_sort(arr):
    n = len(arr)

    # Base case: if n = 0 return [] and if n = 1 return [num1]
    if(n==0 or n==1):
        return arr

    # Divide the array into 2 halves and call merge_sort on both
    sorted_arr1 = merge_sort(arr[:n//2])
    sorted_arr2 = merge_sort(arr[n//2:])

    # Call merge to merge both sorted_arr1 and sorted_arr2
    sorted_arr = merge(sorted_arr1, sorted_arr2)

    return sorted_arr

def merge(sorted_arr1, sorted_arr2):
    n1 = len(sorted_arr1)
    n2 = len(sorted_arr2)

    i=0; j=0
    merged_arr = []
    while(i<n1 and j<n2):
        if(sorted_arr1[i] <= sorted_arr2[j]):
            merged_arr.append(sorted_arr1[i])
            i += 1
        else:
            merged_arr.append(sorted_arr2[j])
            j += 1

    # If elements are left in sorted_arr1
    if(i<n1):
        merged_arr += sorted_arr1[i:]

    # If elements are left in sorted_arr2
    if(j<n2):
        merged_arr += sorted_arr2[j:]


    return merged_arr
```

```python
print("Example-1: merge_sort(arr)")
arr = [4, 3, 2, 10, 12, 1, 5, 6]
print(merge_sort(arr))

print("Example-2: merge_sort(arr)")
arr = [12, 11, 13, 5, 6]
print(merge_sort(arr))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 15:07:11 ~/Personal/Notebooks/Algorithms/2. Searching & Sorting $ python3 3_merge_sort.py
Example-1: merge_sort(arr)
[1, 2, 3, 4, 5, 6, 10, 12]
Example-2: merge_sort(arr)
[5, 6, 11, 12, 13]
```

Complexity:

- **Time:-** Worst Case:**O(nlogn)** Average Case: **O(nlogn)**
- **Auxilliary Space: O(n)**

Notes:

- **Algorithmic Paradigm:** Divide and Conquer Approach
- **Sorting In Place:** No
- **Stable:** Yes

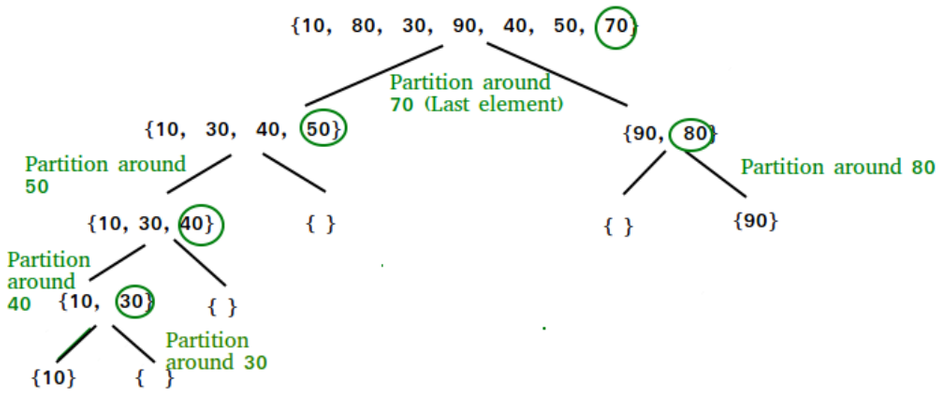
Uses:

- Merge Sort is useful for sorting linked lists in O(nLogn) time.
- Inversion Count Problem
- Used in External Sorting

Algo-3: Quick Sort***

What is Quick Sort ?

- Like Merge Sort, QuickSort is a Divide and Conquer algorithm.
- It picks an element as pivot and partitions the given array around the picked pivot.
- **There are many different versions of quickSort that pick pivot in different ways.**
    - Always pick first element as pivot.
    - Always pick last element as pivot (implemented below)
    - Pick a random element as pivot.
    - Pick median as pivot.
- The key process in quickSort is `partition()`. Target of partitions is to put pivot x of array at its correct position in sorted array and put all elements smaller than x before x, and all elements greater than x after x. All this should be done in linear time.



Algorithm:

- **quick_sort( ):**
    - If size is 0 or 1 rerun the array.
    - call the **partition( )** function to get the partition_index.
    - Now do: **quick_sort(arr[:p_index]) + [arr[p_index]] + quick_sort(arr[p_index+1:])**
    - Return the sorted array
- **partition ( ):**
    - Take last element as pivot.
    - Initialize i =0
    - Start from first element and check if it is lesser than pivot:
        - swap the current element with ith element and increase i.
    - Once every element is done return partition_index as i-1

Implementation:

```python
def quick_sort(arr):
    n = len(arr)

    if( n==0 or n==1):
        return arr

    p_index = partition(arr)

    # Sort the left side of array till partition_index
    # Sort the right side of array from partition_index till the end
    # Return the combined array
    sorted_arr = quick_sort(arr[:p_index]) + [arr[p_index]] + quick_sort(arr[p_index+1:])

    return sorted_arr


def partition(arr):
    n = len(arr)
    # Select the last element as pivot
    pivot = arr[-1]
    i = 0
    for j in range(n):
        if(arr[j] <= pivot):
            arr[i], arr[j] = arr[j], arr[i]
            i+=1

    # Index of pivot as pivot is also considered in swap hence i-1
    return i-1


print("Example-1: quick_sort(arr)")
arr = [10, 80, 30, 90, 70, 85, 75, 95, 40, 50, 70]
print(quick_sort(arr))

print("Example-2: quick_sort(arr)")
arr = [4, 3, 2, 10, 12, 1, 5, 6]
print(quick_sort(arr))

print("Example-3: quick_sort(arr)")
arr = [12, 11, 13, 5, 6]
print(quick_sort(arr))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 16:46:10 ~/Personal/Notebooks/Algorithms/2. Searching & Sorting $ python 4_quick_sort.py
Example-1: quick_sort(arr)
[10, 30, 40, 50, 70, 70, 75, 80, 85, 90, 95]
Example-2: quick_sort(arr)
[1, 2, 3, 4, 5, 6, 10, 12]
Example-3: quick_sort(arr)
[5, 6, 11, 12, 13]
```

Complexity:

- **Time:-** Worst Case:**O(n^2)** Average Case: **O(nlogn)**
- **Auxilliary Space: O(1)**

Notes:

- **Algorithmic Paradigm:** Divide and Conquer Approach
- **Sorting In Place:** Yes
- **Stable:** No

Uses:

- Quick Sort is preferred over MergeSort for sorting Arrays.
- MergeSort is preferred over QuickSort for Linked Lists

Algo-4: Heap Sort***

Heap sort is a comparison based sorting technique based on **Binary Heap** data structure.

It is similar to selection sort where we first find the maximum element and place the maximum element at the end and repeat the same process for remaining element.

Algorithm:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap.
3. Replace it with the last item of the heap followed by reducing the size of heap by 1.
4. Finally, heapify the root of tree.
5. Repeat above steps while size of heap is greater than 1.

Implementation

```
def heap_sort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n//2, -1, -1):
        max_heapify(arr, i, n)
```

```
    # One by one move larger elements to end and decrease the size
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        max_heapify(arr, 0, i)

    return arr


def max_heapify(arr, i, N):
    left_child = 2*i+1
    right_child = 2*i+2

    # Find the largest of left_child, right_child and parent which is i.
    if (left_child < N and arr[left_child] > arr[i]):
        largest = left_child
    else:
        largest = i

    if (right_child < N and arr[right_child] > arr[largest]):
         largest = right_child

    # If Parent is not largest, swap and apply max_heapify on child to propagate it down
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        max_heapify(arr, largest, N)


print("Example-1: heap_sort(arr)")
arr = [10, 80, 30, 90, 70, 85, 75, 95, 40, 50, 70]
print(heap_sort(arr))

print("Example-2: heap_sort(arr)")
arr = [4, 3, 2, 10, 12, 1, 5, 6]
print(heap_sort(arr))

print("Example-3: heap_sort(arr)")
arr = [12, 11, 13, 5, 6]
print(heap_sort(arr))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 17:26:04 ~/Personal/Notebooks/Algorithms/2. Searching & Sorting $ python3 5_heap_sort.py
Example-1: heap_sort(arr)
[10, 30, 40, 50, 70, 70, 75, 80, 85, 90, 95]
Example-2: heap_sort(arr)
[1, 2, 3, 4, 5, 6, 10, 12]
Example-3: heap_sort(arr)
[5, 6, 11, 12, 13]
```

Complexity:

- **Time:-** Worst Case:**O(nlogn)** Average Case: **O(nlogn)**
- **Auxilliary Space: O(1)**

Notes:

- **Algorithmic Paradigm:** Choose Min or Max Approach
- **Sorting In Place:** Yes
- **Stable:** No

Uses:

- Sort a nearly sorted (or K sorted) array
- k largest(or smallest) elements in an array

Standard Sorting Algorithms Problems

1. Sorting Log Files

Problem:

You have an array of logs. Each log is a space delimited string of words.

For each log, the first word in each log is an alphanumeric identifier. Then, either:

- Each word after the identifier will consist only of lowercase letters, or;
- Each word after the identifier will consist only of digits.

We will call these two varieties of logs letter-logs and digit-logs. It is guaranteed that each log has at least one word after its identifier.

Reorder the logs so that all of the letter-logs come before any digit-log. The letter-logs are ordered lexicographically ignoring identifier, with the identifier used in case of ties. The digit-logs should be put in their original order.

Return the final order of the logs.

Approach:

- We can split each log by first space to get identifier and log_data.
- We just need to sort the letter logs by log_data and if log_data matches then on the basis of identifier using custom comparator.
- We also need to ignore the digit logs.
- So we need to write out custom comparator in such a way that it ignores digit logs and sort letter logs by log_data and identifier for ties.

Implementation:

Code:

```
class Solution:
    def reorderLogFiles(self, logs):
        logs.sort(key=self._get_key)
        return logs

    def _get_key(self, log):
        identifier, log_data = log.split(" ", 1)
        if (log_data[0].isalpha()):
            return (0, log_data, identifier)
        else:
            return (1, )


logs = ["dig1 8 1 5 1", "let1 art can", "dig2 3 6", "let2 own kit dig", "let3 art zero"]
print(Solution().reorderLogFiles(logs))
```

Output:

```
['let1 art can', 'let3 art zero', 'let2 own kit dig', 'dig1 8 1 5 1', 'dig2 3 6']
```

Complexity:

- ***Time: O(M*N*LogN)***
  - Here, N is total number of logs and M is the max length of single.
  - N*LogN to sort n items and then each log item is of length M hence, M*N*LognN.
- ***Space: O(M*N)***
  - Sorting of n items take N space and each item is of length M, hence M*N.

2. Nuts & Bolts (Lock & Key) Problem***

Problem:

Given a set of n nuts of different sizes and n bolts of different sizes. There is a one-one mapping between nuts and bolts.

Match nuts and bolts efficiently.

Constraints:

- Comparison of a nut to another nut or a bolt to another bolt is not allowed.
- It means nut can only be compared with bolt and bolt can only be compared with nut to see which one is bigger/smaller.

Other way of asking this problem:

Given a box with locks and keys where one lock can be opened by one key in the box. We need to match the pair.

Example Representation:

Nuts represented as array of character: ***char nuts[]*** = {'@', '#', '$', '%', '^', '&'}

Bolts represented as array of character: ***char bolts[]*** = {'$', '%', '&', '^', '@', '#'}

Approach-1: Brute Force

- Start with the first bolt and compare it with each nut until we find a match.
- **Time Complexity: $O(n^2)$**

Approach-2: Quick Sort

- Perform a partition by picking last element of bolts array as pivot, rearrange the array of nuts and returns the partition index **'i'** such that all nuts smaller than **nuts[i]** are on the left side and all nuts greater than **nuts[i]** are on the right side.
- Next using the nuts[i] we can partition the array of bolts, partitioning operations can easily be implemented in O(n) and this operation also makes nuts and bolts array nicely partitioned.
- Now apply this partitioning recursively on the left and right sub-array of nuts and bolts.
- As we apply partitioning on nuts and bolts both so the total time complexity will be Θ(2*nlogn) = **Θ(nlogn) on average**.
- Here for the sake of simplicity we have chosen last element always as pivot. We can do randomized quick sort too.

Implementation

```
def nuts_bolts_match(nuts, bolts, low, high):
    if low < high:
        # Set last character of bolts for nuts partition.
        pivot = partition(nuts, low, high, bolts[high])

        # Now using the partition index of nuts set pivot for bolts partition
        partition(bolts, low, high, nuts[pivot])
```

```
        # Recur for [low...pivot-1] & [pivot+1...high] for nuts and bolts array.
        nuts_bolts_match(nuts, bolts, low, pivot-1)
        nuts_bolts_match(nuts, bolts, pivot+1, high)


def partition(arr, low, high, pivot):
    i = low
    while(i < high):
        if arr[i] < pivot:
            arr[low], arr[i] = arr[i], arr[low]
            low += 1
        elif arr[i] == pivot:
            arr[high], arr[i] = arr[i], arr[high]
            i -= 1

        i += 1

    arr[low], arr[high] = arr[high], arr[low]

    return low


print("Example-1: Nuts and Bolts Problem")
nuts = ['@', '#', '$', '%', '^', '&']
bolts = ['$', '%', '&', '^', '@', '#']
nuts_bolts_match(nuts, bolts, 0, 5)
print(nuts)
print(bolts)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 23:19:09 ~/Personal/Notebooks/Algorithms/13. Miscellaneous $ python3 1_nuts_bolts.py
Example-1: Nuts and Bolts Problem
['#', '$', '%', '&', '@', '^']
['#', '$', '%', '&', '@', '^']
```

Complexity:

- **Time:-** Average Case: **O(nlogn)**
- **Auxilliary Space: O(1)**

Selection Algorithms

Why Selection Algorithms ?

- Many a times, we need to select find kth smallest or largest element from a given list of elments.
- We can make use of selection algorithms to find that elements.

Famous Selection Algorithms

- QuickSelect
- PartialSelectionSort

Algo-1: QuickSelect Algorithm

What is QuickSelect Algorithm ?

- Quickselect is a selection algorithm to find the kth smallest element in an unsorted list.
- It is closely related to the quicksort sorting algorithm and has **O(N)** average time complexity.
- Like quicksort, it was developed by **Tony Hoare**, and is also known as ***Hoare's selection algorithm***.
- Like quicksort, it is efficient in practice and has good average-case performance, but has poor worst-case performance.
- We can use **Randomized QuickSelect** to further improve the performance.

```
Input: arr[] = {7, 10, 4, 3, 20, 15}
           k = 3
Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}
           k = 4
Output: 10
```

Algorithm:

- First find the **pivot point or partition_index (pat_index)** using **quick sort's partition method**.
- **pat_index or pivot** is a position that partitions the list into two parts: every element on the left is less than the pivot and every element on the right is more than the pivot).
- The algorithm recurs only for the part that contains the k-th smallest element.

- If the index of the partitioned element (pivot) is more than k, then the algorithm recurs for the left part.
- If the index (pivot) is same as k, then we have found the k-th smallest element and it is returned.
- If index is less than k, then the algorithm recurs for the right part.

Partition Method Implementation

```
def partition(arr, low, high):
    # Below 2 lines of code is just to make it Randomized QuickSelect
    rand_index = random.randint(low, high)
    arr[high], arr[rand_index] = arr[rand_index], arr[high]

    pivot = arr[high]

    for i in range(low, high):
        if arr[i] < pivot:
            arr[i], arr[low] = arr[low], arr[i]
            low += 1

    arr[low], arr[high] = arr[high], arr[low]
    return low
```

Kth Smallest Selection Implementation

```
def find_kth_smallest(arr, low, high, k):
    if (low <= high):
        pat_index = partition(arr, low, high)
        if (pat_index == k - 1):
            return arr[k - 1]
        elif (pat_index > k - 1):
            return find_kth_smallest(arr, low, pat_index - 1, k)
        else:
            return find_kth_smallest(arr, pat_index + 1, high, k)
```

Complexity:

- **Time: O(N)** in the average case, **O(N²)** in the worst case.
- **Space: O(1)**

Notes:

- Here the array is everytime split into two parts. If that would be a quicksort algorithm, one would proceed recursively to use quicksort for the both parts that would result in **O(NlogN)** time complexity.
- Here there is no need to deal with both parts since now we know in which part to search for kth smallest element, and that reduces average time complexity to **O(N)**.
- As in QuickSort if everytime we get the worst pivot we will end up sorting the complete array and the worst case time complexity will be **O(N²)** which in practice is almost impossible as we are using Randomized Version of it.

Standard Selection Algorithms Problems

1. Find Kth Smallest Element in Unsorted Array

Problem:

Given a list of elements in unsorted arr, find the kth smallest element.

```
Examples:
Input: [7, 10, 4, 3, 20, 15] and k = 3
Output: 7

Input: [10, 4, 5, 8, 6, 11, 26] and k = 3
Output: 6
```

Approach-1: Use Sorting

- We can sort and get the kth smallest element simply by getting element at (k-1)th index.
- **Time Complexity: O(NLogN)**
- **Space Compelxity: O(1)**

Approach-2: Use Max-Heap of Size K

- We can use a max-heap of size of K and put K elements into it first.
- Then start process all remaining elements, compare it with the max element at top of heap and ignore all elements that is greater or equal to element at top and only insert element that is smaller thatn elment at top of heap into heap.
- **Time Complexity: O(NLogK)**
- **Space Compelxity: O(K)**

Approach-3: Use Randomized QuickSelect

- We can use QuickSelect learned above to solve it.
- **Time complexity: O(N)** in the average case

Implementation

Code:

```
import random


def find_kth_smallest(arr, low, high, k):
    if (low <= high):
        p_index = partition(arr, low, high)
        if (p_index == k - 1):
            return arr[p_index]
        elif (p_index > k - 1):
            return find_kth_smallest(arr, low, p_index-1, k)
        else:
            return find_kth_smallest(arr, p_index + 1, high, k)

    return - 1


def partition(arr, low, high):
    rand_index = random.randint(low, high)
    arr[rand_index], arr[high] = arr[high], arr[rand_index]

    pivot = arr[high]

    for i in range(low, high):
        if (arr[i] < pivot):
            arr[low], arr[i] = arr[i], arr[low]
            low += 1

    arr[low], arr[high] = arr[high], arr[low]

    return low


print(find_kth_smallest([7, 10, 4, 3, 20, 15], 0, 5, 3))
print(find_kth_smallest([10, 4, 5, 8, 6, 11, 26], 0, 6, 3))
print(find_kth_smallest([10, 4, 5, 8, 6, 11, 26], 0, 6, 9))
print(find_kth_smallest([5], 0, 0, 1))
```

Output:

```
7
6
-1
5
```

Complexity:

- **Time: O(N)** in the average case, **O(N^2)** in the worst case.
- **Space: O(1)**

2. Find Kth Largest Element in Unsorted Array

Problem:

Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

```
# Examples:
Input: [3,2,1,5,6,4] and k = 2
Output: 5

Input: [3,2,3,1,2,4,5,5,6] and k = 4
Output: 4
```

Approach: Use Randomized QuickSelect

- Earlier we use to find the Kth smallest element, here we need to find Kth Largest element.
- To find Kth largest element we can simply find the (N-K+1)th smallest element.
- **Time complexity: O(N)** in the average case

Implementation:

Code:

```python
from typing import List
import random


class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        return self.find_kth_smallest_element(nums, 0, len(nums) - 1, len(nums) - k + 1)

    def find_kth_smallest_element(self, arr, low, high, k):
        if (low <= high):
            pat_index = self.partition(arr, low, high)
            if (pat_index == k - 1):
                return arr[k - 1]
            elif (pat_index > k - 1):
                return self.find_kth_smallest_element(arr, low, pat_index - 1, k)
            else:
                return self.find_kth_smallest_element(arr, pat_index + 1, high, k)

    def partition(self, arr, low, high):
        rand_index = random.randint(low, high)
        arr[high], arr[rand_index] = arr[rand_index], arr[high]

        pivot = arr[high]

        for i in range(low, high):
            if arr[i] < pivot:
                arr[i], arr[low] = arr[low], arr[i]
```

```
                low += 1

        arr[low], arr[high] = arr[high], arr[low]
        return low


s = Solution()
assert s.findKthLargest([1, 2, 3, 4, 5], 1) == 5
assert s.findKthLargest([1, 2, 3, 4, 5], 1) == 5
assert s.findKthLargest([1, 2, 3, 4, 5], 5) == 1
assert s.findKthLargest([2, 2, 2, 2, 2], 2) == 2
assert s.findKthLargest([3, 2, 1, 5, 6, 4], 2) == 5
assert s.findKthLargest([3, 2, 3, 1, 2, 4, 5, 5, 6], 4) == 4
```

Complexity:

- ***Time: O(N)*** in the average case, **O(N²)** in the worst case.
- ***Space: O(1)***

3. Find K Closest Points to Origin

Problem:

Given list of points on the plane. Find the k closest points to the origin. You may return the answer in any order.

Examples:

```
Input: points = [[1,3],[-2,2]], K = 1
Output: [[-2,2]]
Explanation:
The distance between (1, 3) and the origin is sqrt(10).
The distance between (-2, 2) and the origin is sqrt(8).
Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.
We only want the closest K = 1 points from the origin, so the answer is just [[-2,2]].

Input: points = [[3,3],[5,-1],[-2,4]], K = 2
Output: [[3,3],[-2,4]]
(The answer [[-2,4],[3,3]] would also be accepted.)
```

Approach-1: Use Sorting

- Sort the points by distance, then take the closest K points.
- **Time Complexity:** O(NLogN), where N is the length of points.

- **Space Complexity:** O(N).

Approach-2: Use Randomized QuickSelect

- Use Randomized QuickSelect to find the Kth smallest element with comparison b/w two elements done on the calculated distance.
- Then return all element upto K coz all these element will be smaller than all elements to the right of K.
- As we only need to give K smallest element in any order simply returning them will suffice.
- If we are asked to return in sorted order sort all elements upto K and return them.
- **Time complexity: O(N)** in the average case

Implementation:

Code:

```
from typing import List
import random


class Solution:
    def kClosest(self, points: List[List[int]], K: int) -> List[List[int]]:
        return self._kClosestUtil(points, 0, len(points)-1, K)

    def _kClosestUtil(self, points, low, high, K):
        if low <= high:
            partition_index = self._partition(points, low, high)
            if (partition_index == K - 1):
                return points[:K]
            elif (partition_index > K - 1):
                return self._kClosestUtil(points, low, partition_index - 1, K)
            else:
                return self._kClosestUtil(points, partition_index + 1, high, K)

    def _dist_from_origin(self, point):
        return point[0]*point[0] + point[1]*point[1]

    def _partition(self, points, low, high):
        rand_index = random.randint(low, high)
        points[rand_index], points[high] = points[high], points[rand_index]

        pivot = points[high]
        pivot_dist_origin = self._dist_from_origin(pivot)
        for i in range(low, high):
            if (self._dist_from_origin(points[i]) < pivot_dist_origin):
                points[low], points[i] = points[i], points[low]
                low += 1

        points[low], points[high] = points[high], points[low]
        return low
```

```
s = Solution()
print(s.kClosest([[1, 3], [-2, 2]], 1))
print(s.kClosest([[3, 3], [5, -1], [-2, 4]], 2))
print(s.kClosest([[3, 3], [5, -1], [-2, 4]], 3))
```

Output:

```
[[-2, 2]]
[[3, 3], [-2, 4]]
[[3, 3], [-2, 4], [5, -1]]
```

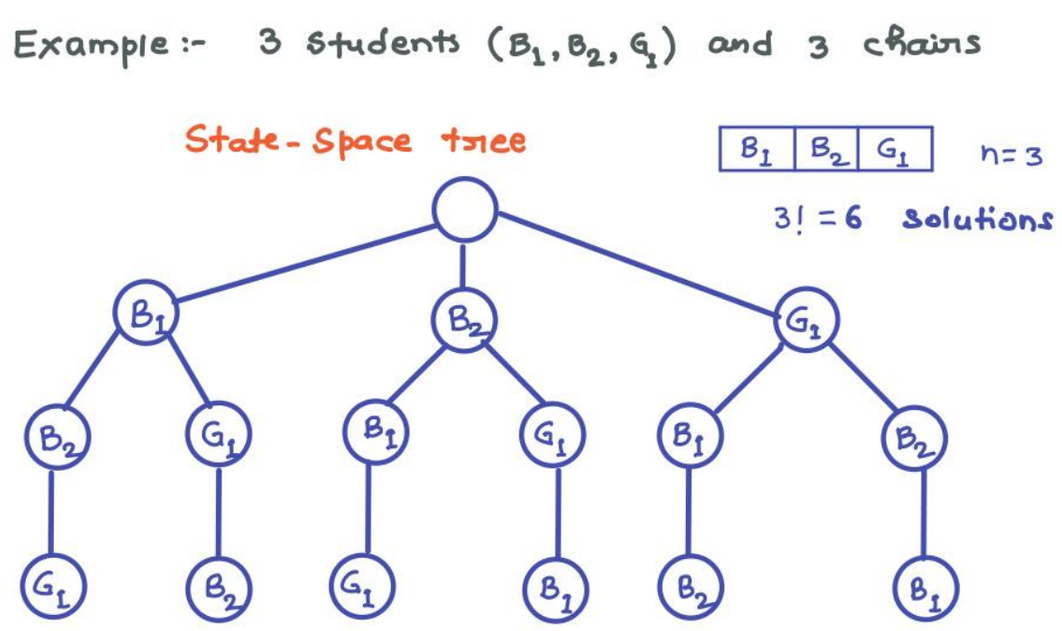
Complexity:

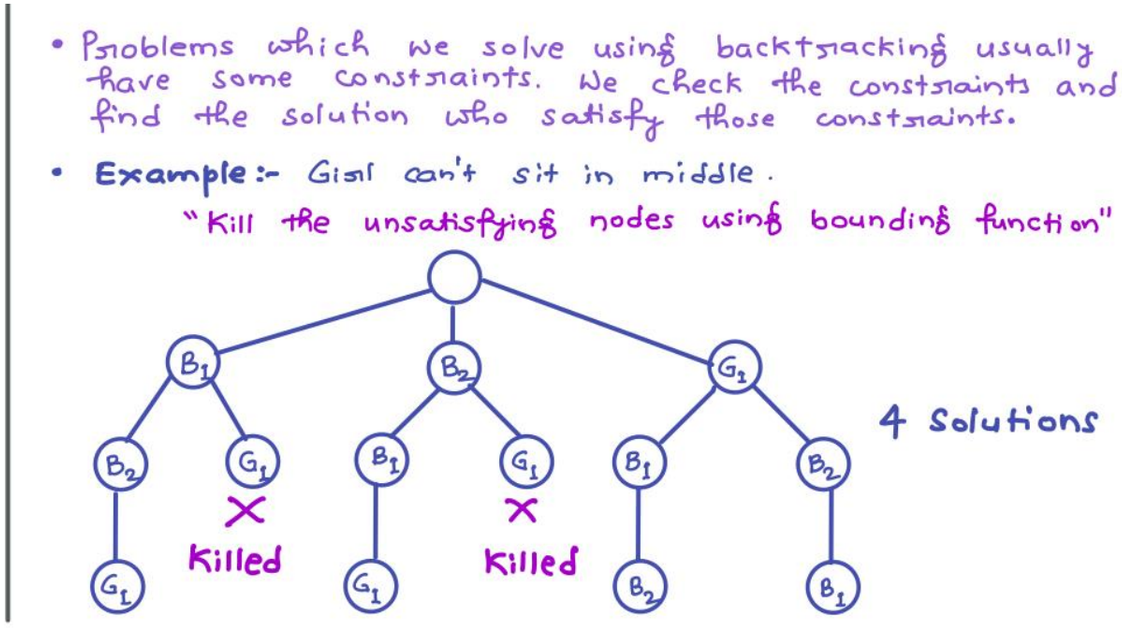
- ***Time: O(N)*** in the average case, **O(N^2)** in the worst case.
- ***Space: O(1)***

Recursion and Backtrack Approach

What is Backtracking Approach ?

- It is one of problem solving strategy and it uses **Brute-Force Approach**.
- For any given problem try out all possible solutions and pick-up desired solutions.
- We also follow this in Dynamic Programming approach but there we solve optimization problem.
- Backtracking is not optimization problem, it is used when we have multiple solutions and we want all those solutions.
- We can find all the solution and it can be represented in the form of a **solution tree** also k/a **state-space tree**.





- **Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that "works".**

Problems which are typically solved using backtracking technique have following property in common:

- These problems can only be solved by trying every possible configuration and each configuration is tried only once.
- A Naive solution for these problems is to try all configurations and output a configuration that follows given problem constraints.
- Backtracking works in incremental way and is an optimization over the Naive solution where all possible configurations are generated and tried.

Notes:-

- There is 1 more strategy that follows Brute-Force approach and that is **Branch and Bound**.
- It also generates state-space tree.
- **But backtracking uses DFS and Branch and Bound uses BFS approach.**

Standard Backtracking Problems:

- All Permutations
- Knight tour
- N-Queen
- Rat in Maze
- Subset-Sum

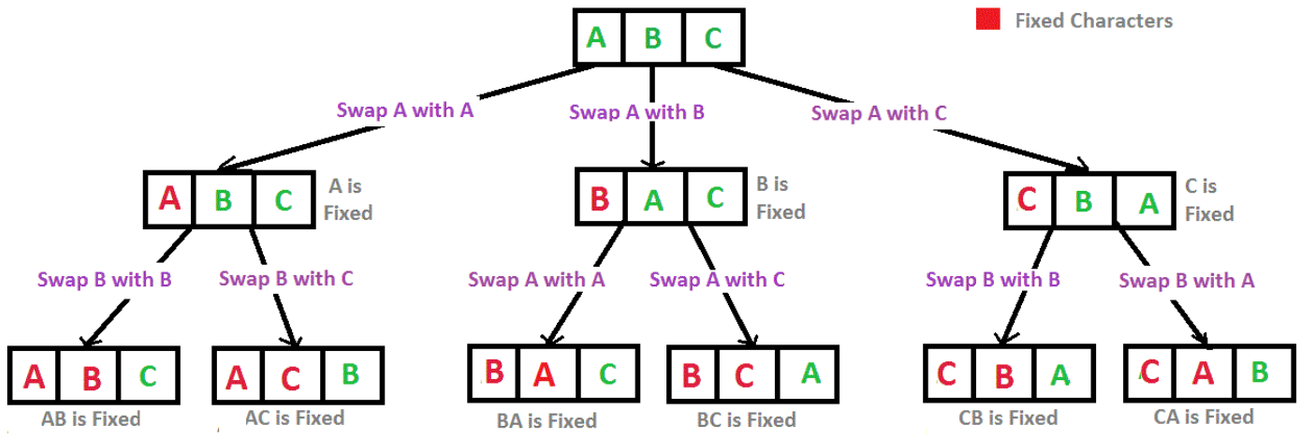
- Sudoku
- Graph-Coloring

Standard Backtracking Problems

1. All Permutations of a Given String***

Problem:

- A permutation, also called an “arrangement number” or “order,” is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself.
- A string of length n has **n!** permutation.
- **Permutations of string ABC:** ABC ACB BAC BCA CBA CAB



Recursion Tree for Permutations of String "ABC"

Algorithm:

- start from left = 0 and right = n.
- if (left==right):
    - print(string) and return
- for i in range(left, right):

- swap the characters of string of (i and left)
- call the function recursively with (left+1, right)
- swap the characters again of (i and left) for backtracking.

Implementation:

```
def generate_permutations_util(string, left, right):
    if(left == right):
        print("{}".format("".join(string)))
        return

    for i in range(left, right):
        string[left], string[i] = string[i], string[left]
        generate_permutations_util(string, left+1, right)
        string[left], string[i] = string[i], string[left]    # Backtrack


def generate_permutations(string):
    n = len(string)
    generate_permutations_util(list(string), 0, n)



print("Example-1:")
generate_permutations("ABC")

print("\nExample-2")
generate_permutations("ABCD")


# Complexity:
#     • Time: O(n*n!) :- There are n! permutations and it requires O(n) time to print a permutation.
#     • Auxilliary Space: O(1)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 14:24:47 ~/Personal/Notebooks/Algorithms/3. Backtracking $ python3 1_all_permutations.py
Example-1:
ABC
ACB
CAB
CBA
ABC
ACB
```

Complexity:

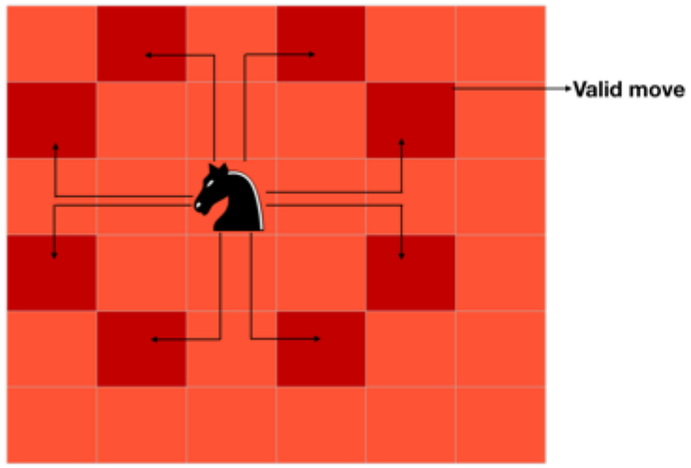
- **Time: O(n*n!)** :- There are n! permutations and it requires O(n) time to print a permutation.

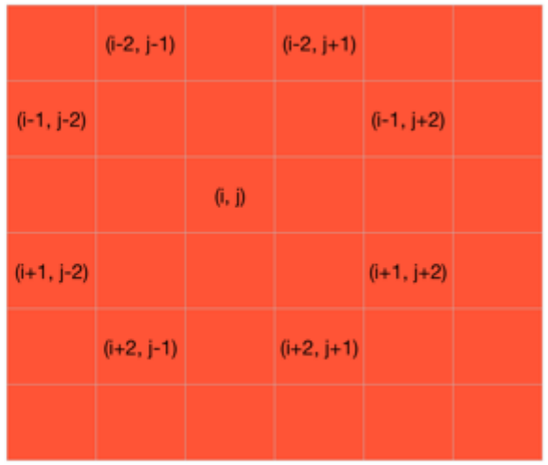
- **Auxilliary Space: O(1)**

2. Knight's Tour Problem***

Problem:

The knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once.





Naive Approach:

The Naive Algorithm is to generate all tours one by one and check if the generated tour satisfies the constraints.

```
while there are untried tours {
   generate the next tour
   if this tour covers all squares {
      print this path;
   }
}
```

Backtracking Approach:

- It works in an incremental way to attack problems.
- Typically, we start from an empty solution vector and one by one add items.
- Meaning of item varies from problem to problem, in context of Knight's tour problem, an item is a Knight's move.

- When we add an item, we check if adding the current item violates the problem constraint, if it does then we remove the item and try other alternatives.
- If none of the alternatives work out then we go to previous stage and remove the item added in the previous stage.
- If we reach the initial stage back then we say that no solution exists.
- If adding an item doesn't violate constraints then we recursively add items one by one.
- If the solution vector becomes complete then we print the solution.

Implementation:

```
POSSIBLE_MOVES = [(-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2,1)]

def valid_move(x, y, tour_matrix):
    return (x>=0 and x<N and y>=0 and y<N and tour_matrix[x][y]==-1)


def knight_tour_util(current_x, current_y, move_number, tour_matrix):
    if(move_number == N*N):
        return True

    # Try all possible moves
    for x_move, y_move in POSSIBLE_MOVES:
        next_x = current_x + x_move
        next_y = current_y + y_move
        if(valid_move(next_x, next_y, tour_matrix)):
            tour_matrix[next_x][next_y] = move_number
            if(knight_tour_util(next_x, next_y, move_number+1, tour_matrix) == True):
                return True
            else:
                tour_matrix[next_x][next_y] = -1  # Backtrack
    return False


def knight_tour():
    # Create a 2-D Matrix(N*N)
    tour_matrix = [[-1]*N for _ in range(N)]

    #  Knight is initially at the first block
    tour_matrix[0][0]  = 0

    if(knight_tour_util(0, 0, 1, tour_matrix) == True):
        for i in range(N):
            for j in range(N):
                print("{0: =2d}".format(tour_matrix[i][j]), end=" ")
            print()
        print()
    else:
        print("No solution exist")



print("Knight Tour Example-1: 4*4 Matrix")
N = 4
knight_tour()
```

```python
print("\nKnight Tour Example-2: 5*5 Matrix")
N = 5
knight_tour()

print("\nKnight Tour Example-3: 8*8 Matrix")
N = 8
knight_tour()


# Complexity:
#     • Time: O(8^(n^2)) :- There are N*N i.e., N^2 cells in the board and we have a maximum of 8 choices to make from a cell.
#     • Auxilliary Space: O(N^2)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 18:05:52 ~/Personal/Notebooks/Algorithms/3. Backtracking $ python3 2_knight_tour.py
Knight Tour Example-1: 4*4 Matrix
No solution exist

Knight Tour Example-2: 5*5 Matrix
 0 17  4  9  2
 5 10  1 18 13
16 21 12  3  8
11  6 23 14 19
22 15 20  7 24


Knight Tour Example-3: 8*8 Matrix
 0 11  8  5  2 13 16 19
 9  6  1 12 17 20  3 14
30 27 10  7  4 15 18 21
63 24 31 28 35 22 47 44
32 29 26 23 48 45 36 57
25 62 51 34 39 56 43 46
52 33 60 49 54 41 58 37
61 50 53 40 59 38 55 42
```

Complexity:

- **Time: $O(8^{N^2})$:-** There are N*N i.e., N^2 cells in the board and we have a maximum of 8 choices to make from a cell.
- **Auxilliary Space: $O(N^2)$:-** Need to create a solution matrix of N*N.

Notes:

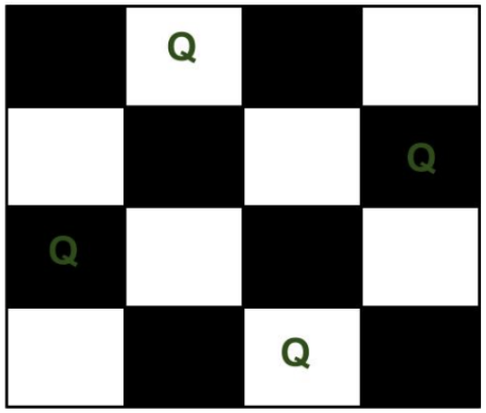
- Backtracking is not the best solution for the Knight's tour problem
- Other better solutions: **Warnsdorff's algorithm for Knight's problem.**

3. N-Queen Problem***

Problem:

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other.

Example: 4 Queen problem.



Approach:

- The idea is to place queens one by one in different columns, starting from the leftmost column.
- When we place a queen in a column, we check for clashes with already placed queens.
- In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution.
- If we do not find such a row due to clashes then we backtrack and return false.

Algorithm:

1. Start in the leftmost column

2. If all queens are placed: **return true**
3. Try all rows in the current column. Do following for every tried row.
    - a) If the queen can be placed safely in this row then mark this [row,column] as part of the solution and recursively check if placing queen here leads to a solution.
    - b) If placing the queen in [row, column] leads to a solution then **return true**.
    - c) If placing queen doesn't lead to a solution then unmark this [row,column] (Backtrack) and go to step (a) to try other rows.
4. If all rows have been tried and nothing worked, **return false** to trigger backtracking.

Implementation:

```
# To check if a queen can be placed on board[row][col].
# Note that this function is called when "col" number of queens are already placed in columns from 0 to col -1.
# So we need to check only left side for attacking queens.
def is_safe(board, row, col):
    safe = True
    # Check this row on left side.
    for j in range(col):
        if(board[row][j] == 1):
            safe = False
            break

    # Check upper diagonal on left side
    i = row; j=col
    while(i>=0 and j>=0):
        if(board[i][j]==1):
            safe = False
            break
        i-=1; j-=1

    # Check lower diagonal on left side
    i = row; j = col
    while(i<N and j>=0 and safe):
        if(board[i][j]==1):
            safe = False
            break
        i+=1; j-=1

    return safe


def n_queen_util(board, col):
    # Base case: If all queens are placed then return true
    if(col == N):
        return True

    # Consider this column and try placing this queen in all rows one by one
    for i in range(N):
        if(is_safe(board, i, col)):
            # Place this queen in board[i][col]
            board[i][col] = 1
            # Recur to place rest of the queens
            if(n_queen_util(board, col+1) == True):
                return True
```

```
            else:
                board[i][col] = 0    # Backtrack

    # If the queen can not be placed in any row in this colum col  then return false
    return False


def n_queen():
    board = [[0]*N for i in range(N)]

    if(n_queen_util(board, 0) == True):
        for i in range(N):
            print(board[i])
    else:
        print("No solution exists")



print("N-Queen Example-1: 3*3 Matrix")
N = 3
n_queen()

print("\nN-Queen Example-2: 4*4 Matrix")
N = 4
n_queen()

print("\nN-Queen Example-3: 5*5 Matrix")
N = 5
n_queen()

print("\nN-Queen Example-4: 8*8 Matrix")
N = 8
n_queen()


# Complexity:
#     • Time:
#     • Auxilliary Space:
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 03:25:22 ~/Personal/Notebooks/Algorithms/3. Backtracking $ python3 3_n_queen.py
N-Queen Example-1: 3*3 Matrix
No solution exists

N-Queen Example-2: 4*4 Matrix
[0, 0, 1, 0]
[1, 0, 0, 0]
[0, 0, 0, 1]
[0, 1, 0, 0]

N-Queen Example-3: 5*5 Matrix
[1, 0, 0, 0, 0]
[0, 0, 0, 1, 0]
[0, 1, 0, 0, 0]
[0, 0, 0, 0, 1]
[0, 0, 1, 0, 0]

N-Queen Example-4: 8*8 Matrix
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
```

Complexity:

- **Time: O(N!)**
- **Auxilliary Space: O(N^2)**:- Need to create a board matrix of N*N.

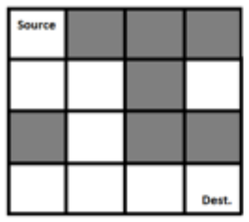
4. Rat in Maze***

Problem:

- A Maze is given as N*N binary matrix of blocks where **source** block is the upper left most block i.e., `maze[0][0]` and **destination** block is lower rightmost block i.e., `maze[N-1][N-1]`.
- A rat starts from source and has to reach the destination.
- The rat can move only in two directions: forward and down.
- In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination.
- Note that this is a simple version of the typical Maze problem.
- **Example:** A more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

Example Maze:

Maze

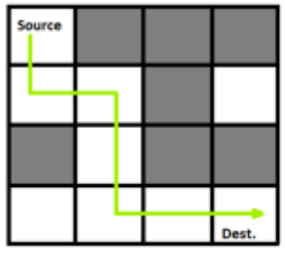


Grey blocks are dead end (value=0)

Following is binary matrix representation of the above maze.

```
[1, 0, 0, 0]
[1, 1, 0, 1]
[0, 1, 0, 0]
[1, 1, 1, 1]
```

Following is a maze with highlighted solution path.



Following is the solution matrix (output of program) for the above input matrix.

```
[1, 0, 0, 0]
[1, 1, 0, 0]
[0, 1, 0, 0]
[0, 1, 1, 1]

All enteries in solution path are marked as 1.
```

Algorithm:

- **If destination is reached:**
  - print the solution matrix
- **Else:**
  - a) Mark current cell in solution matrix as 1.
  - b) Move forward in the horizontal direction and recursively check if this move leads to a solution.
  - c) If the move chosen in the above step doesn't lead to a solution then move down and check if this move leads to a solution.
  - d) If none of the above solutions works then unmark this cell as 0(BACKTRACK) and return false.

Implementation:

```
def rat_in_maze_util(solution_maze, x, y):
    if(x == N-1 and y == N-1):
        return True

    # Make a move in forward direction if it is_safe to move
    if(given_maze[x][y+1] == 1):
        solution_maze[x][y+1] = 1
        if(rat_in_maze_util(solution_maze, x, y+1) == True):
            return True
        else:
            solution_maze[x][y+1] = 0    # Backtrack

    # Make a move in downward direction if it is_safe to move
    if(given_maze[x+1][y] == 1):
        solution_maze[x+1][y] = 1
        if(rat_in_maze_util(solution_maze, x+1, y) == True):
            return True
        else:
            solution_maze[x][y+1] = 0    # Backtrack

    return False


def rat_in_maze():
    solution_maze = [[0]*N for i in range(N)]
    solution_maze[0][0] = 1

    if(rat_in_maze_util(solution_maze, 0, 0)):
        for i in range(N):
            print(solution_maze[i])
    else:
        print("No Solution exist.")



print("Rat in Maze Example-1: 4*4 Matrix")
N = 4
given_maze = [ [1, 0, 0, 0],
               [1, 1, 0, 1],
               [0, 1, 0, 0],
               [1, 1, 1, 1] ]
rat_in_maze()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 18:41:18 ~/Personal/Notebooks/Algorithms/3. Backtracking $ python3 4_rat_in_maze.py
Rat in Maze Example-1: 4*4 Matrix
[1, 0, 0, 0]
[1, 1, 0, 0]
[0, 1, 0, 0]
[0, 1, 1, 1]
```

Complexity:

- **Time:** Every block will have 2 directional choices (Forward & Backward). Hence 222*. . .(n times), so $\mathbf{2^n}$.
- **Auxilliary Space: O(N^2):-** Need to create a maze matrix of N*N.

5. Subset Sum Problem

Problem:

- Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K.
- We are considering the set contains non-negative values.
- It is assumed that the input set is unique (no duplicates are presented).
- The problem is **NP-Complete**, while it is easy to confirm whether a proposed solution is valid, it may be difficult to determine in the first place whether any solution exists.
- There exists a **DP solution** to this problem which gives **pseudo-polynomial** time and hence considered **weakly NP-Complete**.

  Example-1: Set = [10, 7, 5, 18, 12, 20, 15] K = 35 then,

  Answer: [10, 7, 18] or [20, 15]

  Example-2: Set = [15, 22, 14, 26, 32, 9, 16, 8] K = 53 then,

  Answer: [15, 22, 16] or [32, 9, 16]

Implementation:

```
def subset_sum(given_set, num):
    if num < 1 or len(given_set) == 0:
        return False
```

```
    if num == given_set[0]:
        return [given_set[0]]

    with_v = subset_sum(given_set[1:], num-given_set[0])
    if with_v:
        return [given_set[0]] + with_v
    else:
        return subset_sum(given_set[1:], num)



print("Subset Sum Example-1:")
given_set = [10, 7, 5, 18, 12, 20, 15]
print(subset_sum(given_set, 35))

print("Subset Sum Example-2:")
given_set = [15, 22, 14, 26, 32, 9, 16, 8]
print(subset_sum(given_set, 53))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 00:44:04 ~/Personal/Notebooks/Algorithms/3. Backtracking $ python3 5_subset_sum.py
Subset Sum Example-1:
[10, 7, 18]
Subset Sum Example-2:
[15, 22, 16]
```

Complexity:

- **Time:** Every number will be either picked or not picked (2choices). Hence 222*. . .(n times), so $\mathbf{2^n}$.
- **Auxilliary Space: O(N)**

6. Sudoku***

Problem:

Given a partially filled 9×9 2D array `grid[9][9]`, the goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and subgrid of size 3×3 contains exactly one instance of the digits from 1 to 9.

3		6	5		8	4		
5	2							
	8	7					3	1
		3		1			8	
9	5		8	6 9	3	6		5
1	3					2	5	
							7	4
		5	2		6	3		

Backtracking Approach:

- Like all other Backtracking problems, we can solve Sudoku by one by one assigning numbers to empty cells.
- Before assigning a number, we check whether it is safe to assign i.e. check that the same number is not present in the current row, current column and current 3X3 subgrid.
- After checking for safety, we assign the number, and recursively check whether this assignment leads to a solution or not.
- If the assignment doesn't lead to a solution, then we try next number for the current empty cell.
- And if none of the number (1 to 9) leads to a solution, we return false.

Algorithm:

- Find row, col of an unassigned cell and If there is none, return true
- For digits from 1 to 9
  - a) If there is no conflict for digit at row, col assign digit to row, col and recursively try fill in rest of grid
  - b) If recursion successful, return true
  - c) Else, remove digit and try another
- If all digits have been tried and nothing worked, return false

Implementation

```
import math

# get_base_box gives start row or col of the mini_box
# like if row = 2 then base_row = 0 and if row = 8 then base_row = 6
def get_base_box(index, k):
    return int(index/k)*k
```

```python
def find_unassigned_box(matrix):
    n = len(matrix[0])
    for row in range(n):
        for col in range(n):
            if(matrix[row][col]==0):
                return (True, row, col)

    return (False, None, None)


def is_safe(matrix, row, col, num):
    n = len(matrix[0])
    status = True
    # Check if the row is safe
    for i in range(n):
        if(matrix[row][i] == num):
            status = False
            break

    # Check if the column is safe
    for i in range(n):
        if(matrix[i][col] == num):
            status = False
            break

    # Check if the individual 3*3 box is safe as n = 9 hence k = 3
    k = int(math.sqrt(n))
    base_row = get_base_box(row, k)
    base_col = get_base_box(col, k)
    for i in range(k):
        for j in range(k):
            if(matrix[i+base_row][j+base_col] == num):
                status = False
                break

    return status


def solve_sudoku(matrix):
    n = len(matrix[0])
    # Check for unassigned box and return status, if status is True also return row and col of that box
    found, row, col = find_unassigned_box(matrix)

    # If No unassigned box found, we are done
    if(not found):
        return True

    # Consider digits 1 to 9
    for num in range(1, n+1):
        # Check if it is safe to put this number
        if(is_safe(matrix, row, col, num)):
            # Put this number
            matrix[row][col] = num
            # Recur to check if it leads to solution
            if(solve_sudoku(matrix) == True):
                return True
            else:
```

```
                matrix[row][col] = 0   # Backtrack

    return False



print("Sudoku Example-1:")
matrix = [[3,0,6,5,0,8,4,0,0],
          [5,2,0,0,0,0,0,0,0],
          [0,8,7,0,0,0,0,3,1],
          [0,0,3,0,1,0,0,8,0],
          [9,0,0,8,6,3,0,0,5],
          [0,5,0,0,9,0,6,0,0],
          [1,3,0,0,0,0,2,5,0],
          [0,0,0,0,0,0,0,7,4],
          [0,0,5,2,0,6,3,0,0]]

n = len(matrix[0])
if(solve_sudoku(matrix)):
    for i in range(n):
        for j in range(n):
            print(matrix[i][j], end=" ")
        print()
else:
    print("No solution exists")
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 03:14:38 ~/Personal/Notebooks/Algorithms/3. Backtracking $ python3 6_sudoku.py
Sudoku Example-1:
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
```

Complexity:

- **Time:**
- **Auxilliary Space: $O(N^2)$**

7. Tower of Hanoi

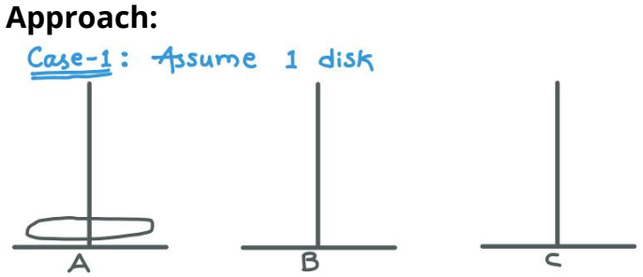
Problem:

3 Towers given, and one of tower has all the disks kept in increasing order of size from top to bottom.

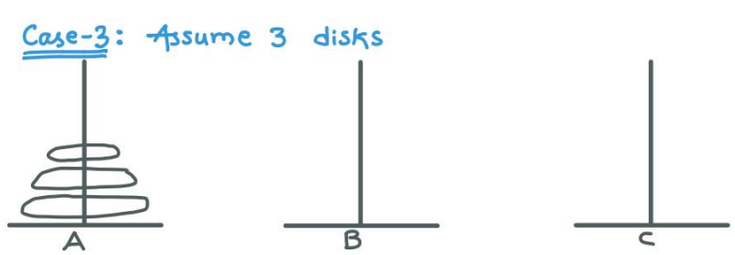
We have to move all those disks to another tower with below conditions:

- Only one disk can be moved at a time.
- At no point of time a larger disk can be kept on smaller.

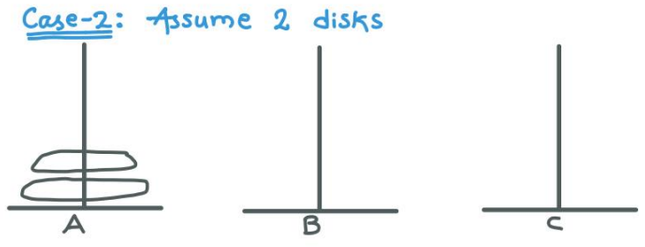
Approach:



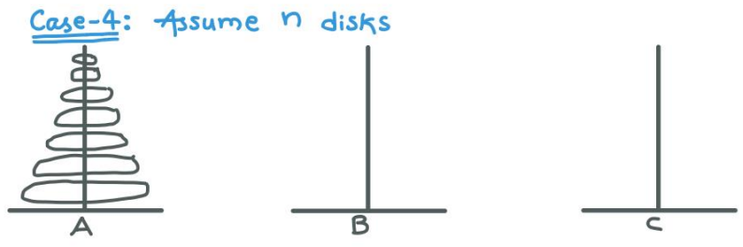
- Move 1 disk from A to C



- Move 2 disks from A to B using C (Recursively)
- Move 1 disk from A to C
- Move 2 disks from B to C using A (Recursively)



- Move 1 disk from A to B using C
- Move 1 disk from A to C
- Move 1 disk from B to C using A



- Move n-1 disks from A to B using C (Recursively)
- Move 1 disk from A to C
- Move n-1 disks from B to C using A (Recursively)

Implementation

```
def tower_of_hanoi(n, source, auxiliary, destination):
    if(n>0):
        # Move n-1 disks from source to auxilliary using destination
        tower_of_hanoi(n-1, source, destination, auxiliary)

        # Move that 1 disk now from source to destination
        print("Move 1 disk from {} to {}".format(source, destination))

        # Move rest n-1 from auxiliary to destination using source
        tower_of_hanoi(n-1, auxiliary, source, destination)



print("Tower of Hanoi Example-1")
tower_of_hanoi(3, "A", "B", "C")

print("\nTower of Hanoi Example-2")
tower_of_hanoi(4, "A", "B", "C")
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 04:41:10 ~/Personal/Notebooks/Algorithms/3. Backtracking $ python3 7_tower_of_hanoi.py
Tower of Hanoi Example-1
Move 1 disk from A to C
Move 1 disk from A to B
Move 1 disk from C to B
Move 1 disk from A to C
Move 1 disk from B to A
Move 1 disk from B to C
Move 1 disk from A to C
```

Complexity:

- **Time: $O(2^n)$**
- **Auxilliary Space: O(1)**

Greedy Approach

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.

Where Greedy Algorithms are used ?

- Greedy algorithms are used for optimization problems.
- **An optimization problem can be solved using Greedy if the problem has the following property:**
  - At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.
- If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem.
- Because the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming.
- But Greedy algorithms cannot always be applied.
- **Example:-** Fractional Knapsack problem can be solved using Greedy, but 0-1 Knapsack can't be solved using Greedy.

Standard Greedy Algorithm Problems:

1. **Kruskal's Minimum Spanning Tree (MST):**
   - Create a MST by picking edges one by one.
   - The Greedy Choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.
2. **Prim's Minimum Spanning Tree (MST):**
   - Create a MST by picking edges one by one.
   - Maintain two sets: a set of the vertices already included in MST and the set of the vertices not yet included.
   - The Greedy Choice is to pick the smallest weight edge that connects the two sets.
3. **Dijkstra's Shortest Path (SPT):**
   - The Dijkstra's algorithm is very similar to Prim's algorithm.
   - The shortest path tree is built up, edge by edge.
   - Maintain two sets: a set of the vertices already included in the tree and the set of the vertices not yet included.
   - The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.
4. **Huffman Coding:**
   - A loss-less compression technique.
   - It assigns variable-length bit codes to different characters.
   - The Greedy Choice is to assign least bit length code to the most frequent character.
5. **Approximation for Hard optimization problems:**
   - The greedy algorithms are sometimes also used to get an approximation for Hard optimization problems.
   - Example: **Traveling Salesman Problem** is a **NP-Hard problem**.
   - A Greedy choice for this problem is to pick the nearest unvisited city from the current city at every step.
   - This solutions don't always produce the best optimal solution but can be used to get an approximately optimal solution.

Standard Greedy Algorithm Problems

1. Activity Selection Problem***

Problem:

Given n activities with their start and finish times.

Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Example-1: Consider the following 3 activities sorted by finish time.

start[] = {10, 12, 20}

finish[] = {20, 25, 30}

Output: (10, 20) (20, 30)

Example-2: Consider the following 6 activities sorted by finish time.

start[] = {3 , 0 , 5 , 8 , 5, 1}

finish[] = {4 , 6 , 9 , 9 , 7, 2}

Output: (1, 2) (3, 4) (5, 7) (8, 9)

Algorithm:

- Sort the activities by the finish time in increasing order and preferably if finish times are same then start time in decreasing order.
- Just start picking the activities from start.
- Make sure that the start time of next activity is greater or equal to finish time of previous activity.

Implementation

```
def activity_selection(start, finish):
    activities = []
    n = len(start)
    for i in range(n):
        activities.append((start[i], finish[i]))

    # Sort the activities by finish time in increasing order and if finish times same then start time in decreasing order
    activities.sort(key=lambda k: (k[1], -k[0]))

    print("Selected Activities:")
    prev_finish = 0
    for current_start, current_finish in activities:
        if(current_start >= prev_finish):
            print("({}, {})".format(current_start, current_finish))
            prev_finish = current_finish



print("Example-1:")
```

```
start =  [10, 12, 20]
finish = [20, 25, 30]
activity_selection(start, finish)

print("\nExample-2:")
start =  [3 , 0 , 5 , 8 , 5, 1]
finish = [4 , 6 , 9 , 9 , 7, 2]
activity_selection(start, finish)


# Time : O(n log n) if input activities not sorted.
#      : O(n) if input activities are sorted.
# Auxilliary Space: O(n) :-> Creating a new array but can be done in O(1)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 21:13:16 ~/Personal/Notebooks/Algorithms/4. Greedy Algorithms $ python3 1_activity_selection.py
Example-1:
Selected Activities:
(10, 20)
(20, 30)

Example-2:
Selected Activities:
(1, 2)
(3, 4)
(5, 7)
(8, 9)
```

Complexity:

- **Time:**
  - **O(n log n):** if input activities not sorted.
  - **O(n):** if input activities are sorted.
- **Auxilliary Space: O(n)** : Creating a new array but can be done in O(1)

2. Minimum Number of Platforms Required***

Problem:

Given arrival and departure times of all trains that reach a railway station, find the minimum number of platforms required for the railway station so that no train waits.

Example:

Input:

arr[] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}

dep[] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}

Output: 3

Algorithm:

- Sort the arrivals and departures.
- Now check the sequence of Arrival and Departures.
- If arrival is there increase count and if departure decrease count but before decreasing check if count > platforms then platforms = count

Implementation:

```
def min_number_of_platforms_required(arrivals, departures):
    n = len(arrivals)
    arrivals.sort()
    departures.sort()

    i =0; j=0; count = 0; platforms =0
    while(i<n and j<n):
        if(arrivals[i] < departures[j]):
            count += 1
            i += 1
        else:
            if(count > platforms):
                platforms = count
            count -= 1
            j += 1

    print("Minimum Number of Platforms required = {}".format(platforms))



print("Example-1:")
arrivals    = [9.00,  9.40, 9.50,  11.00, 15.00, 18.00]
departures  = [9.10, 12.00, 11.20, 11.30, 19.00, 20.00]
min_number_of_platforms_required(arrivals, departures)


# Time Complexity : O(n log n) if arrivals & departures are not sorted.
#                 : O(n) if arrivals & departures are sorted.
# Auxilliary Space: O(1)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 22:42:11 ~/Personal/Notebooks/Algorithms/4. Greedy Algorithms $ python3 2_min_number_of_platforms.py
Example-1:
Minimum Number of Platforms required = 3
```

Complexity:

- **Time:**
  - **O(n log n):** if input activities not sorted.
  - **O(n):** if input activities are sorted.
- **Auxilliary Space: O(1)**

3. Huffman Coding***

- Huffman coding is a lossless data compression algorithm.
- The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.
- The most frequent character gets the smallest code and the least frequent character gets the largest code.
- The variable-length codes assigned to input characters are Prefix Codes.
- The codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character.
- This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.
- **There are mainly two major parts in Huffman Coding:**
  i. Build a Huffman Tree from input characters.
  ii. Traverse the Huffman Tree and assign codes to characters.

Algorithm to build Huffman Tree:

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes.
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with frequency equal to the sum of the two nodes frequencies.
   - Make the first extracted node as its left child and the other extracted node as its right child.
   - Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Build Huffman Tree Example

Example:

`character` `Frequency`

a ——————————> 5

b ——————————> 9

c ——————————> 12

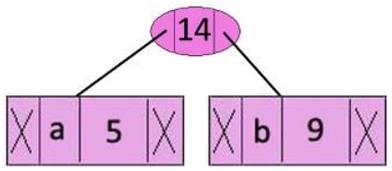
d ——————————> 13

e ——————————> 16

f ——————————> 45

Step-1: Build a min heap that contains **6 nodes** where each node represents root of a tree with single node.

Step-2: Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



Now min heap contains **5 nodes** where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements.

`character` `Frequency`

c ——————————> 12

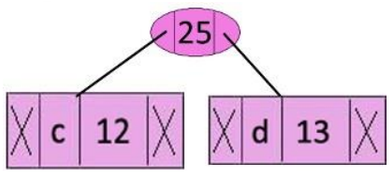
d ——————————> 13

Internal Node ————> 14

e ——————————> 16

f ——————————> 45

Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25.



`character` `Frequency`

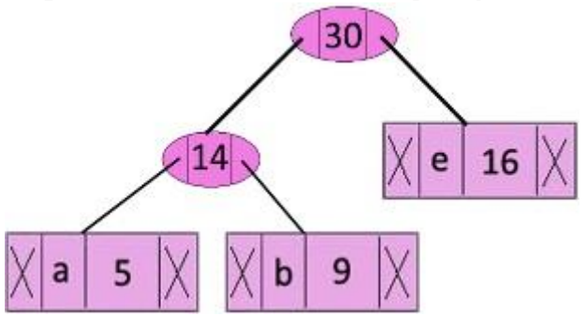
Internal Node ————> 14

e ——————————> 16

Internal Node ————> 25

```
f ——————————> 45
```

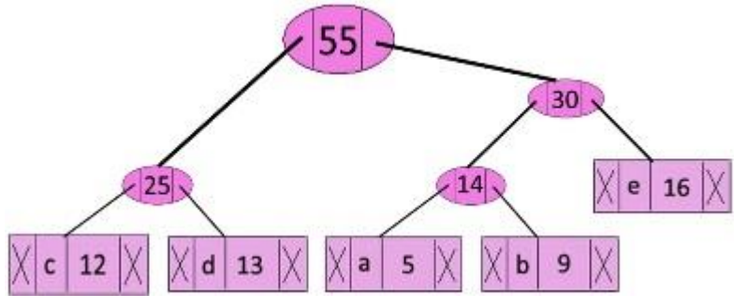
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30.



Now min heap contains **3 nodes**.

```
character Frequency
Internal Node ————>25
Internal Node ————> 30
 f ——————————> 45
```

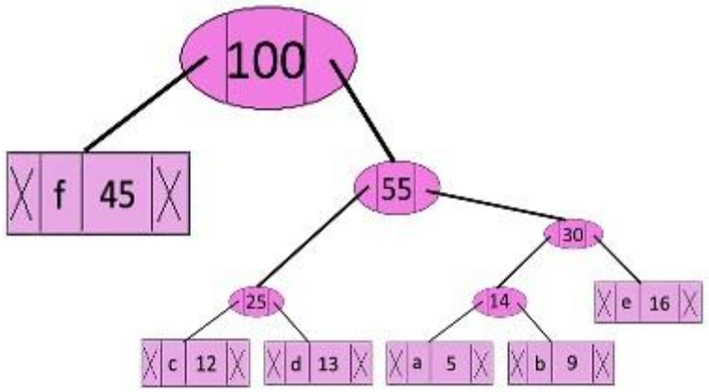
Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55.



Now min heap contains **2 nodes**.

```
character Frequency
f ——————————> 45
Internal Node ——>55
```

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



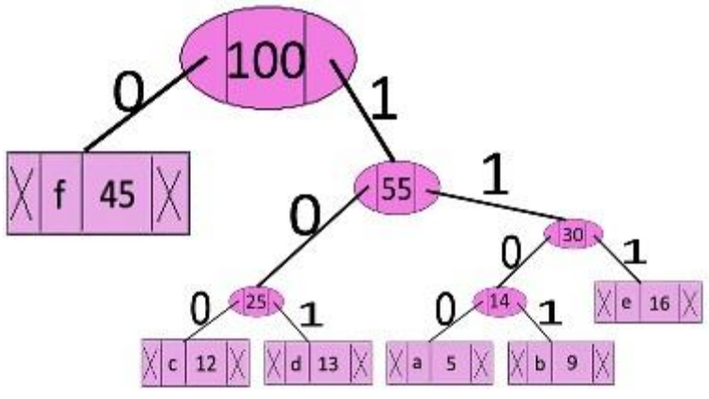
Now min heap contains only **one node**.

```
character Frequency
Internal Node ——->100
```

Since the heap contains only one node, the algorithm stops here.

Algorithm to print codes from Huffman Tree:

1. Traverse the tree formed starting from the root and maintain a code starting with empty string.
2. While moving to the left child, add 0 to the code string and while moving to the right child, add 1 to code string.
3. Print the code when a leaf node is encountered.



The Codes are as follows:

```
character Codes
f ————————> 0
c ————————> 100
```

d ————————> 101

a ————————> 1100

b ————————> 1101

e ————————> 111

Implementation

```
import heapq


class HeapNode:
    def __init__(self, char, frequency):
        self.char = char
        self.frequency = frequency
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.frequency < other.frequency


def build_huffman_tree(chars, frequencies):
    # Create a leaf node for each unique character and build a min heap of all leaf nodes
    n = len(chars)
    custom_heap = []
    for i in range(n):
        heap_node = HeapNode(chars[i], frequencies[i])
        heapq.heappush(custom_heap, heap_node)

    # Extract two nodes with the minimum frequency from the min heap and start merging
    while(len(custom_heap) > 1):
        node_left  = heapq.heappop(custom_heap)
        node_right = heapq.heappop(custom_heap)
        merged_node = HeapNode(None, node_left.frequency+node_right.frequency)
        merged_node.left = node_left
        merged_node.right = node_right
        heapq.heappush(custom_heap, merged_node)

    # Root Node of the Heap
    root_node = heapq.heappop(custom_heap)

    return root_node


def print_huffman_code(node, code):
    if(node.left == None and node.right == None):
        print("{}: {}".format(node.char, code))
        return

    print_huffman_code(node.left, code+"0")
    print_huffman_code(node.right, code+"1")
```

```python
def generate_huffman_code(chars, frequencies):
    root_node = build_huffman_tree(chars, frequencies)
    print_huffman_code(root_node, "")



print("Huffman Codes for characters:")
chars = ['a', 'b', 'c', 'd', 'e', 'f']
frequencies = [5, 9, 12, 13, 16, 45]
generate_huffman_code(chars, frequencies)


# Time Complexity : O(nlogn) where n is the number of unique characters.
#     To extract Min using heappop() takes O(logn) and as there are n nodes, it is called 2*(n - 1) times.
# Auxilliary Space : O(n) :-> Storing n nodes in heap
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 04:28:05 ~/Personal/Notebooks/Algorithms/4. Greedy Algorithms $ python3 3_huffman.py
Huffman Codes for characters:
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Complexity:

- **Time : O(nlogn)** where n is the number of unique characters.
  : To extract Min using heappop() takes O(logn) and as there are n nodes, it is called 2*(n–1) times.
- **Auxilliary Space : O(n)** : Storing n nodes in heap

4. Connect ‘n’ ropes with minimum cost***

Problem:

There are given n ropes of different lengths, we need to connect these ropes into one rope.

The cost to connect two ropes is equal to sum of their lengths, calculate the minimum cost.

Example:

Input: 4 ropes of lengths 4, 3, 2 and 6. ***Output:*** 29

Explanation:

1) First connect ropes of lengths 2 and 3. Now we have three ropes of lengths 4, 6 and 5. 2) Now connect ropes of lengths 4 and 5. Now we have two ropes of lengths 6 and 9. 3) Finally connect the two ropes and all ropes have connected.

Total cost = 5 + 9 + 15 = 29

Approach: Huffman Coding

- Observing closely, we can notice that the lengths of the ropes which are picked first are included more than once in total cost.
- Therefore, the idea is to connect smallest two ropes first and recur for remaining ropes.
- This approach is similar to **Huffman Coding**.
- Put smallest ropes down the tree so that they can be repeated multiple times rather than the longer ropes.
- **Time Complexity: O(nlogn)**

Algorithm:

1. Create a min heap and insert all lengths into the min heap.
2. Do following while number of elements in min heap is not one.
    - a) Extract the minimum and second minimum from min heap
    - b) Add the above two extracted values and insert the added value to the min-heap.
    - c) Maintain a variable for total cost and keep incrementing it by the sum of extracted values.
3. Return the value of this total cost.

Implementation:

```
import heapq

def connect_n_ropes_min_cost(ropes):
    total_cost = 0
    heapq.heapify(ropes)

    while(len(ropes) >= 2):
        new_connected_rope = heapq.heappop(ropes) + heapq.heappop(ropes)
        total_cost += new_connected_rope
        heapq.heappush(ropes, new_connected_rope)

    print(total_cost)


print("Example-1: connect_n_ropes_min_cost([4, 3, 2, 6])")
connect_n_ropes_min_cost([4, 3, 2, 6])
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 14:31:44 ~/Personal/Notebooks/Algorithms/13. Miscellaneous $ python3 5_connect_n_ropes_min_cost.py
Example-1: connect_n_ropes_min_cost([4, 3, 2, 6])
29
```

Complexity:

- **Time: O(nlogn)** : Extract min take O(logn) and need to do it for n times.
- **Auxilliary Space: O(n)**

Dynamic Programming Approach

What is Dynamic Programming Approach ?

- It is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and storing the results of subproblems to avoid computing the same results again.
- Utilizes the fact that the optimal solution to the overall problem depends on the optimal solutions to its subproblems.
- ***Example:*** Fibonacci numbers - We can calculate nth Fibonacci by below eqution

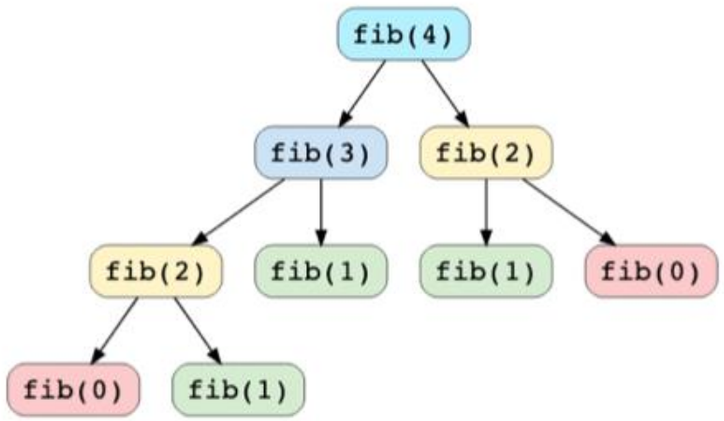
```
fib(n) = fib(n-1) + fib(n-2) for n > 1
```

- Here to solve the overall problem we broke it down to smaller subproblems.

Characteristics of Dynamic Programming

1. Overlapping Subproblems

- Subproblems are smaller version of the original problem.
- Any problem has overlappping subproblems if finding its solution involves solving the same subproblem multiple times.
- Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems.
- Dynamic Programming is mainly used when solutions of same subproblems are needed again and again.
- In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed.
- Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again.
- **Binary Search** is broken down into subproblems but it doesn't have common subproblems, so no sense to store the solutions.

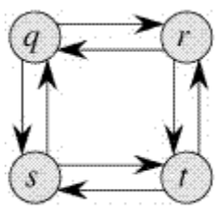


Recursion tree for calculating Fibonacci numbers

- Here we can see overlapping subproblems as **fib(2)** is evaluated twice and **fib(1)** evaluated thrice.

2. Optimal Substructure Property

- Any problem has optimal substructure property if its overall optimal solution can be constructed from optimal solution of subproblems.
- `fib(n)= fib(n-1)+fib(n-2)` - shows that overall problem of size ***n*** is reduced to subproblems of size ***n-1*** and ***n-2***.
- ***Example:-*** The Shortest Path problem has following optimal substructure property:
  - If a node x lies in the shortest path from a source node u to destination node v.
  - Then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v.
  - The standard All Pair Shortest Path algorithms like **Floyd–Warshall** and **Bellman–Ford** are typical examples of Dynamic Programming.
- But the Longest Path problem i.e. longest simple path (path without cycle) between two nodes doesn't have the Optimal Substructure property.
  - There are two longest paths from q to t: q→r→t and q→s→t.
  - Unlike shortest paths, these longest paths do not have the optimal substructure property.
  - *Example:-* The longest path q→r→t is not a combination of longest path from q to r and longest path from r to t.
    - Coz the longest path from q to r is q→s→t→r and the longest path from r to t is r→q→s→t.



Methods to Solve Dynamic Programming Problem

1. Recursion + Memoization (Top-Down Approach)

- Solve bigger problem by recursiverly finding the solution to smaller sub-problems.
- ***Whenever we solve a sub-problem we cache its result to avoid calling it multiple times.***
- This technique of storing the results of already solved subproblems k/a **Memoization**.

2. Iteration + Tabulation (Bottom-Up Approach)

- Tabulation is opposite of Top-Down Approach and avoids recursion.
- In this we solve the problem bottom up (i.e by solving all the related subproblems first).
- ***This is typically done by filling an n-dimensional table.***
- Based on the results in the table, the solution to the top/orignal problem is computed.

Notes :-

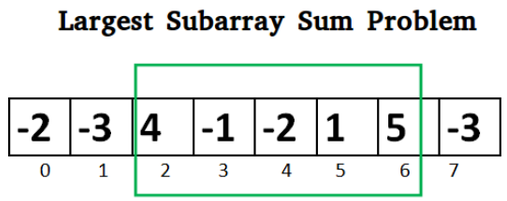
- Both Tabulated and Memoized store the solutions of subproblems.
- In Memoized version, table is filled on demand but in Tabulated version, starting from the first entry, all entries are filled one by one.
- Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version.
- ***Example:-*** Memoized solutionof the LCS problem doesn't necessarily fill all entries.

Standard Dynamic Programming Problems

1. Maximum Sum Subarray***

Problem:

Find the sum of contiguous subarray within a one-dimensional array of numbers which has the largest sum.



4 + (-1) + (-2) + 1 + 5 = 7

Maximum Contiguous Array Sum is 7

Kadane's Algorithm:

- Use Kadane's DP approach.

Implementation

```
def maxSubArray(nums):
    current_sum = max_sum = float("-inf")
    start = end = s = 0

    for i, num in enumerate(nums):
        if (current_sum < 0):
            current_sum = num
            s = i
        else:
            current_sum += num

        if(current_sum >= max_sum):
            max_sum = current_sum
            end = i
            start = s

    print(f"Max : {max_sum} and Subarray : {nums[start:end + 1]}")


maxSubArray([-2, -3, 4, -1, -2, 1, 5, -3])
maxSubArray([-2, 1, -3, 4, -1, 2, 1, -5, 4])
maxSubArray([1])
maxSubArray([0])
maxSubArray([-1])
maxSubArray([-5, -3, -2, -4])
```

Output:

```
Max : 7 and Subarray : [4, -1, -2, 1, 5]
Max : 6 and Subarray : [4, -1, 2, 1]
```

```
Max : 1 and Subarray : [1]
Max : 0 and Subarray : [0]
Max : -1 and Subarray : [-1]
Max : -2 and Subarray : [-2]
```

Complexity:

- **Time: O(n)** :- Loop runs till n
- **Auxilliary Space: O(1)**

2. Subset Sum Problem***

Problem:

Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

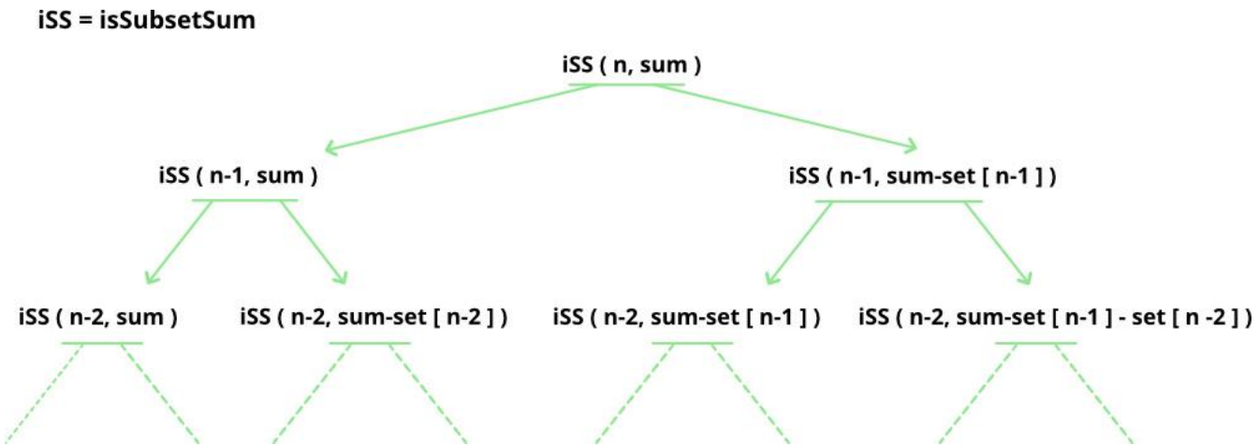
Example:

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 11

Output: True //There is a subset (4, 5, 2) with sum 11.

Recursive Approach:

is_subset_sum(set, n, sum) = is_subset_sum(set, n-1, sum)		is_subset_sum(set, n-1, sum-set[n-1])
- **Base Cases:**
  - is_subset_sum(set, n, sum) = false, if sum > 0 and n == 0
  - is_subset_sum(set, n, sum) = true, if sum == 0



Recursive Implementation:

```
def is_subset_sum_recursive(given_set, n, given_sum):
    if(given_sum==0):
        return True

    if(n==0 and given_sum != 0):
        return False

    return is_subset_sum_recursive(given_set, n-1, given_sum) or
                is_subset_sum_recursive(given_set, n-1, given_sum-given_set[n-1])


print("Example-1: is_subset_sum_recursive(given_set, n, given_sum)")
given_set = [1, 3, 9, 2]
print(is_subset_sum_recursive(given_set, 4, 5))

print("Example-2: is_subset_sum_recursive(given_set, n, given_sum)")
given_set = [3, 34, 4, 12, 5, 2]
print(is_subset_sum_recursive(given_set, 6, 11))

print("Example-3: is_subset_sum_recursive(given_set, n, given_sum)")
given_set = [3, 34, 4, 12, 5, 2]
print(is_subset_sum_recursive(given_set, 6, 13))
```

Output

```
astik.anand@mac-C02XD95ZJG5H 04:21:13 ~/Personal/Notebooks/Algorithms/5. Dynamic Programming $ python3 2_subset_sum.py
Example-1: is_subset_sum_recursive(given_set, n, given_sum)
True
Example-2: is_subset_sum_recursive(given_set, n, given_sum)
True
Example-3: is_subset_sum_recursive(given_set, n, given_sum)
False
```

Complexity:

- **Time: O(2^n)** :- Every number will either be picked or not hence 222..... = **2^n**.
- **Auxilliary Space: O(n)**

Notes:

- The above solution may try all subsets of given set in worst case.
- Therefore time complexity of the above solution is exponential.
- The problem is in-fact **NP-Complete** (There is no known polynomial time solution for this problem).

Dynamic Programming Approach:

- We can solve the problem in **Pseudo-polynomial time** using Dynamic programming.
- Create a boolean 2D table[][] and fill it in bottom up manner.
- The value of `table[i][j]` will be true if there is a subset of set[0..j-1] with sum equal to i., otherwise false.
- Finally, return table`[sum][n]` .

Set : [1 , 3 , 9 , 2]
0 1 2 3 → Set indexes

Sum →

Given Set	0	1	2	3	4	5
0	T	F	F	F	F	F
1	T	T	F	F	F	F
3	T	T	F	T	T	F
9	T	T	F	T	T	F
2	T	T	T	T	T	T

table row indexes

Sum = 0 can be achieved by any set by choosing empty subset. Hence **TRUE** for every subset.

when only **Empty Sub-set { }** is considered all sum will be **False** except Sum = 0

① if table[i-1][j] == True:
    table[i][j] = True

↳ यदि कोई sum, current number को sub-set में include करने के पहले से achievable था, तो अब include करने के बाद भी achievable रहेगा।

② elif j-set[i-1] > 0 and table[i-1][j-set[i-1]] == True:
(Sum, Current number)
    table[i][j] = True

↳ यदि कोई sum, current number को sub-set में include करने से पहले achievable नहीं था, तो अब check करना है कि current_sum − current_number achievable था या नहीं, यदि था तो काम बन गया।

③ else :
    table[i][j] = True

↳ यदि ऊपर का दोनों case से भी True नहीं हो पाया, तो False.

Dynamic Programming Implementation:

```
def is_subset_sum_dp(given_set, n, given_sum):
    table = [[False]*(given_sum+1) for i in range(n+1)]

    # Sum = 0 can be achieved by any subset by taking an empty set
    for i in range(n+1):
        table[i][0] = True

    # With empty set all the sum will be False except sum=0
    for i in range(1, given_sum+1):
        table[0][i] = False

    # Now fill the rest of table
    for i in range(1, n+1):
        for j in range(1, given_sum+1):
            # If earlier (before adding this number in set) sum was possible,
            # the now also it will be possible
            if(table[i-1][j] == True):
                table[i][j] = True
            # If earlier not possible check if current_sum-current_number was possible earlier,
            # if it was then we are done.
            elif(j-given_set[i-1]>=0 and table[i-1][j-given_set[i-1]] == True):
                table[i][j] = True
            # If above 2 cases is not there, then False
            else:
                table[i][j] = False

    return table[n][given_sum]


print("Example-1: is_subset_sum_dp(given_set, n, given_sum)")
given_set = [1, 3, 9, 2]
print(is_subset_sum_dp(given_set, 4, 5))

print("Example-2: is_subset_sum_dp(given_set, n, given_sum)")
given_set = [3, 34, 4, 12, 5, 2]
print(is_subset_sum_dp(given_set, 6, 11))

print("Example-3: is_subset_sum_dp(given_set, n, given_sum)")
given_set = [3, 34, 4, 12, 5, 2]
print(is_subset_sum_dp(given_set, 6, 13))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 04:22:07 ~/Personal/Notebooks/Algorithms/5. Dynamic Programming $ python3 2_subset_sum.py
Example-1: is_subset_sum_dp(given_set, n, given_sum)
True
Example-2: is_subset_sum_dp(given_set, n, given_sum)
True
Example-3: is_subset_sum_dp(given_set, n, given_sum)
False
```

Complexity:

- **Time: O(sum)** :- Loop till max_num in set
- **Auxilliary Space: O(n*sum)**

3. Minimum Jumps to Reach End***

Problem:

Given an array of integers where each element represents the max number of steps that can be made forward from that element.

Write a function to return the minimum number of jumps to reach the end of the array (starting from the first element).

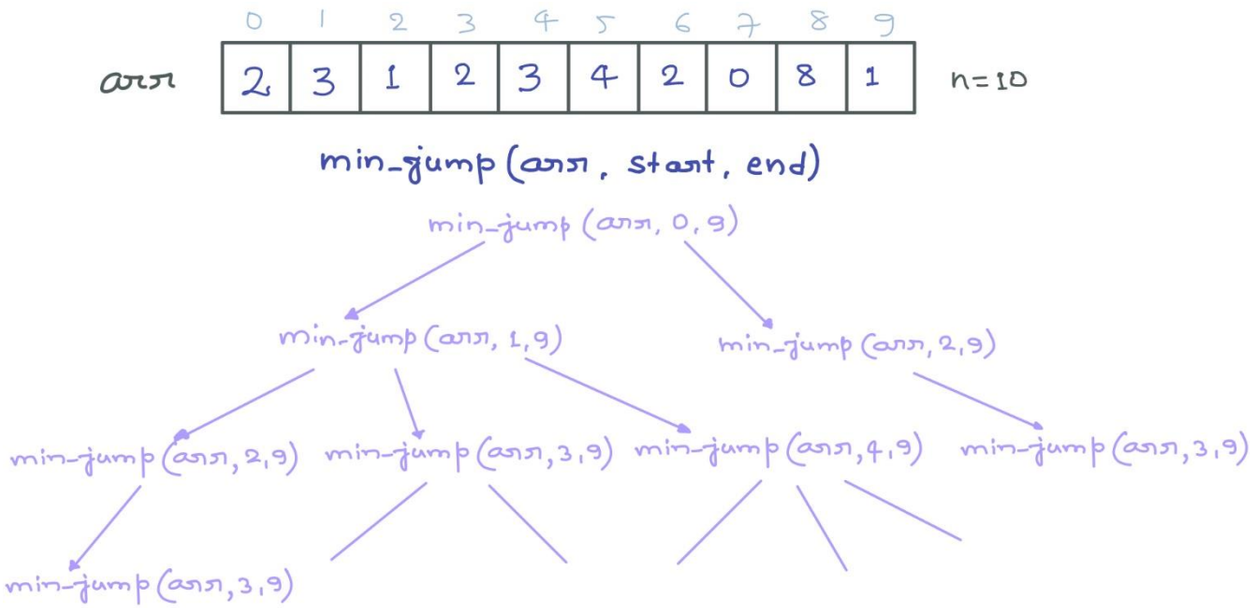
If an element is 0, then cannot move through that element.

Example:

Input: arr[] = {2, 3, 1, 2, 3, 4, 2, 0, 8, 1}

Output: 4 (2-> 3-> 2-> 4-> 1)

Recursive Approach:



Recursive Implementation:

```
import sys

def min_jump_to_reach_end_recursive(arr, start, end):
    # Base case: when start position and end position are same
    if (start == end):
        return 0

    # When nothing is reachable from the given position
    if (arr[start] == 0):
        return sys.maxsize

    # Just check where you can reach from start and then call min_jump from there
    min_jumps = sys.maxsize
    for i in range(1, arr[start]+1):
        next_start = start+i
        if(next_start < n):
            jumps = 1 + min_jump_to_reach_end_recursive(arr, next_start, end)
            if (jumps < min_jumps):
                min_jumps = jumps

    return min_jumps



print("Min Jump Recursive Example-1: min_jump_to_reach_end_recursive(arr, start, end)")
arr = [2, 3, 1, 2, 3, 4, 2, 0, 8, 1]
n = len(arr)
jumps = min_jump_to_reach_end_recursive(arr, 0, n-1)
```

```
if(jumps == sys.maxsize):
    print("Unreachable")
else:
    print("Jumps: {}".format(jumps))

print("\nMin Jump Recursive Example-2: min_jump_to_reach_end_recursive(arr, start, end)")
arr = [2, 3, 1, 2, 3, 2, 1, 0, 8, 1]
n = len(arr)
jumps = min_jump_to_reach_end_recursive(arr, 0, n-1)
if(jumps == sys.maxsize):
    print("Unreachable")
else:
    print("Jumps: {}".format(jumps))
```

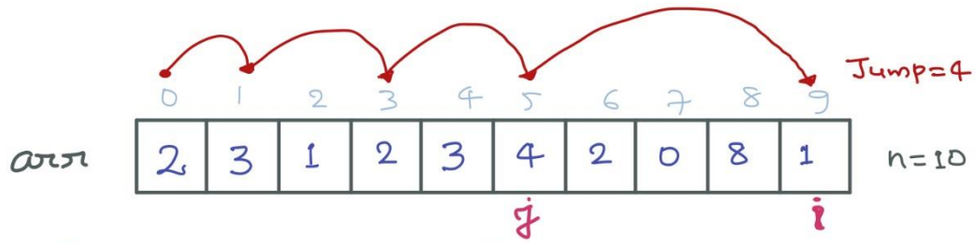
Output:

```
astik.anand@mac-C02XD95ZJG5H 09:57:21 ~/Personal/Notebooks/Algorithms/5. Dynamic Programming $ python3 3_min_jumps_to_reach_end.py
Min Jump Recursive Example-1: min_jump_to_reach_end_recursive(arr, start, end)
Jumps: 4
```

Complexity:

- **Time: O(2^n)** :- Every number will either be picked or not hence 222..... = **2^n**.
- **Auxilliary Space: O(n)**

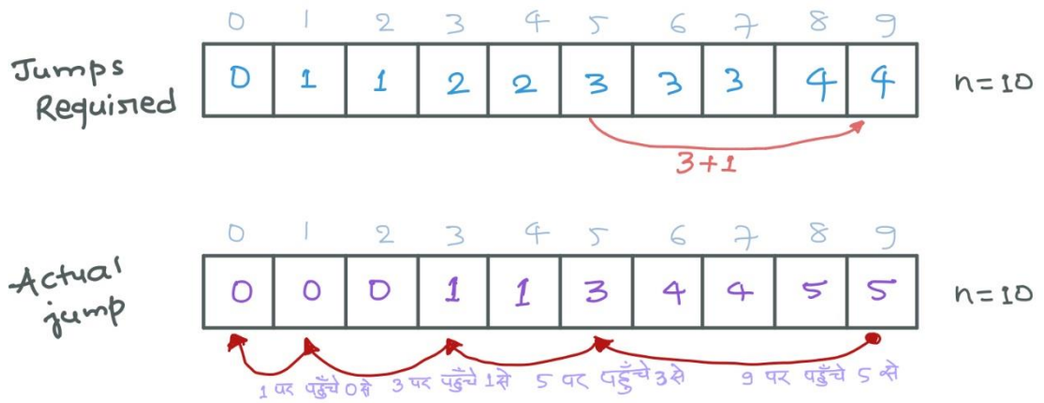
Dynamic Programming Approach:



यदि i तक j से पहुँच सकते हैं; $j + arr[j] \geq i$

तब,

$jump\_to\_reach\_i = min(1 + jump\_to\_reach\_j,\ jump\_to\_reach\_i)$



Dynamic Programming Implementation:

```
def min_jump_to_reach_end_DP(arr):
    n = len(arr)
    # Initialize jumps_required as "infinity" for every element
    jumps_required = [sys.maxsize]*n
    actual_jump = [0]*n

    # First element is reachable by 0 jumps
    jumps_required[0] = 0

    for i in range(1, n):
        for j in range(i):
            # Check if from j we can reach i or not??
            # If j + arr[j] >= i : then we can reach i
            if(j+arr[j] >= i):
                # If i is reachable from j in lesser jumps than earlier update the jumps to reach i
                # Update the actual jump table also
                if(jumps_required[j]+1 < jumps_required[i]):
                    jumps_required[i] = jumps_required[j]+1
                    actual_jump[i] = j
```

```
    # Now print number of jumps and actual jump
    if(jumps_required[n-1] == sys.maxsize):
        print("Unreachable")
    else:
        print("Jumps: {}".format(jumps_required[n-1]))
        k = n-1
        print("Actual Jumps: end", end="")
        while(k>0):
            print("<--{}".format(actual_jump[k]), end="")
            k = actual_jump[k]
        print()



print("Min Jump DP Example-1: min_jump_to_reach_end_DP(arr)")
arr = [2, 3, 1, 2, 3, 4, 2, 0, 8, 1]
min_jump_to_reach_end_DP(arr)

print("\nMin Jump DP Example-2: min_jump_to_reach_end_DP(arr)")
arr = [2, 3, 1, 2, 3, 2, 1, 0, 8, 1]
min_jump_to_reach_end_DP(arr)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 12:17:18 ~/Personal/Notebooks/Algorithms/5. Dynamic Programming $ python3 3_min_jumps_to_reach_end.py
Min Jump DP Example-1: min_jump_to_reach_end_DP(arr)
Jumps: 4
Actual Jumps: end<--5<--3<--1<--0

Min Jump DP Example-2: min_jump_to_reach_end_DP(arr)
Unreachable
```

Complexity:

- **Time: O(n^2)** :- 2 for loops
- **Auxilliary Space: O(n)**

Note:

There exists a O(n) solution to be discussed later.

4. Coin Change - Unique Ways***

Problem:

Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins.

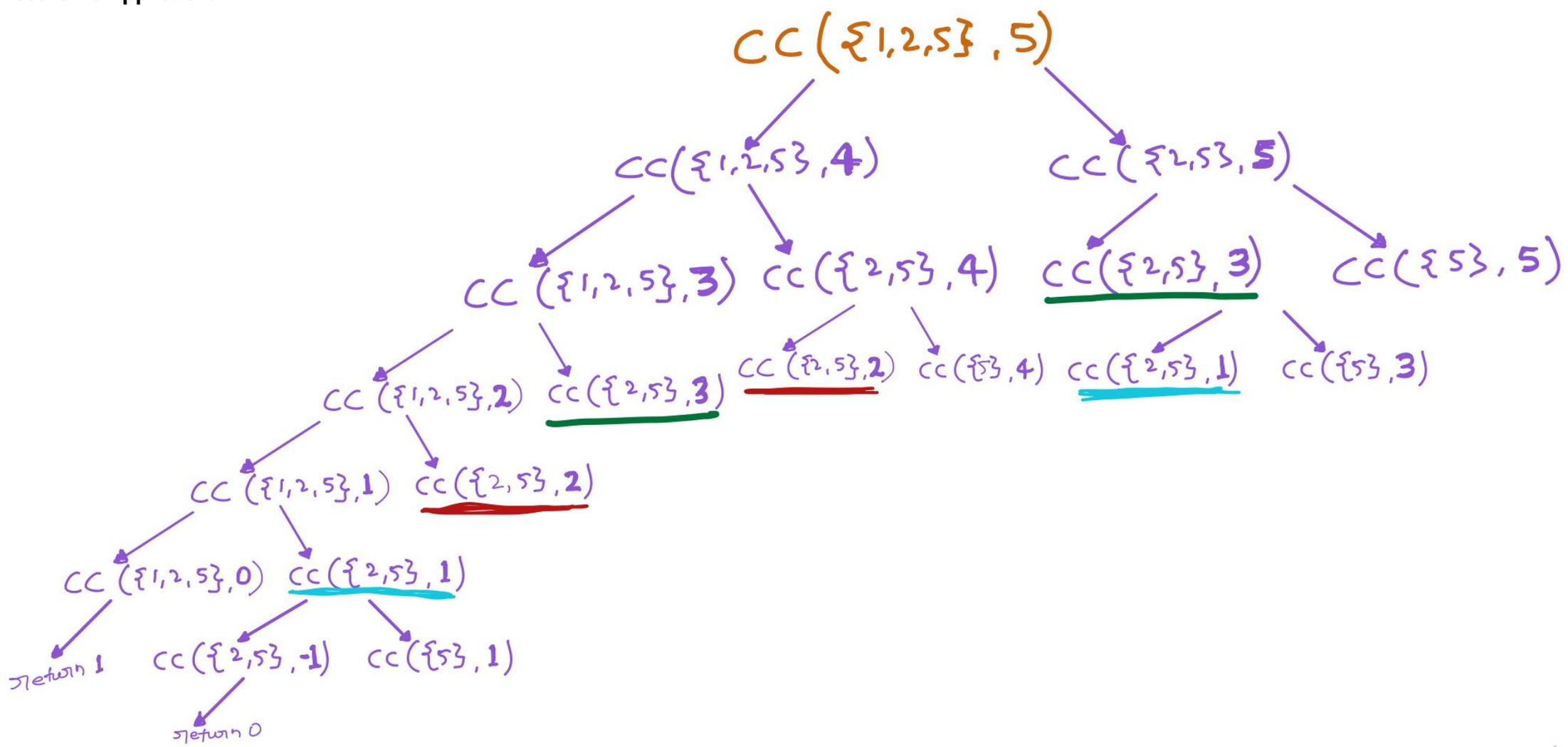
How many unique ways can we make the change? The order of coins doesn't matter.

Example:

Input: N = 5 and S = {1, 2, 5} Output: 4 {1,1,1,1,1}, {1,1,1,2}, {1,2,2}, {5}.

Input: N = 10 and S = {2, 5, 3, 6} Output: 5 {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}

Recursive Approach:



Recursive Implementation:

```
def coin_change_unique_ways_recursive(coins, N):
    m = len(coins)

    # If N is less than 0 then no solution exists
    if (N < 0):
        return 0

    # If N is 0 then 1 solution : Don't include any coin
    if (N == 0):
        return 1

    # If no coins, no solution exist
    if (m == 0):
        return 0

    # Answer is sum of solutions (i)including first coin (ii) excluding first coin
    return coin_change_unique_ways_recursive(coins, N-coins[0]) +
                    coin_change_unique_ways_recursive(coins[1:], N)



print("Example-1: coin_change_unique_ways_recursive(coins, N)")
coins = [1, 2, 5]
print(coin_change_unique_ways_recursive(coins, 5))
print("Example-2: coin_change_unique_ways_recursive(coins, N)")
coins = [2, 5, 3, 6]
print(coin_change_unique_ways_recursive(coins, 10))
```

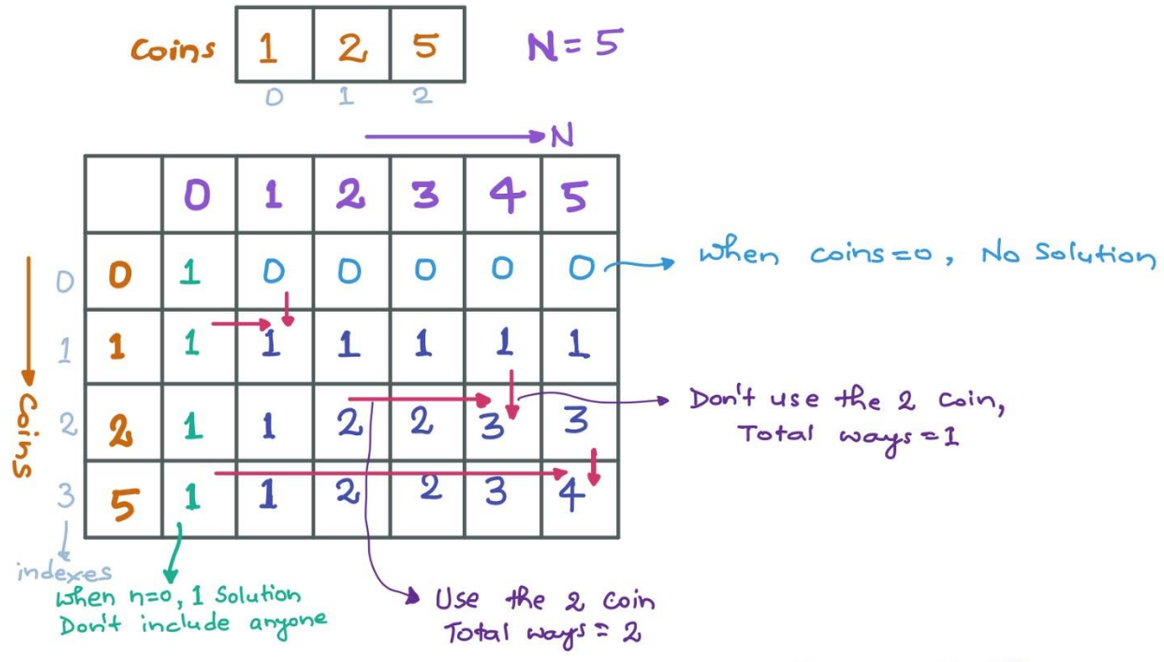
Output:

```
astik.anand@mac-C02XD95ZJG5H 11:19:39 ~/Personal/Notebooks/Algorithms/5. Dynamic Programming $ python3 4_coin_change_unique_ways.py
Example-1: coin_change_unique_ways_recursive(coins, N)
4
Example-2: coin_change_unique_ways_recursive(coins, N)
5
```

Complexity:

- **Time: O(2^n)** :- Every coin will either be picked or not hence 222..... = **2^n**.
- **Auxilliary Space: O(n)**

Dynamic Programming Approach:



Table[i][j] = Table[i-1][j] + Table[i][j - Coins[i-1]]

Don't include ith coin

Include ith coin

Dynamic Programming Implementation:

```
def coin_change_unique_ways_DP(coins, N):
    m = len(coins)
    table = [[0]*(N+1) for i in range(m+1)]

    # when N=0, 1 solution:- Don't include anyone --- Fill first column
    for i in range(m+1):
        table[i][0] = 1

    # When No coins or m = 0 :- No solution possible ---- Fill first row
    # Line can be omitted as table is initialized with 0, just for better understanding
    for j in range(1, N+1):
        table[0][j] = 0

    # Fill rest of the table
    # table[i][j] =        table[i-1][j]         +  table[i][j-coins[i-1]]
    # table[i][j] = (Not including current coin) + (Including current coin)
    for i in range(1, m+1):
        for j in range(1, N+1):
            # Can't include current coin
            if(j-coins[i-1] < 0):
```

```
                table[i][j] = table[i-1][j]
            else:
                table[i][j] = table[i-1][j] + table[i][j-coins[i-1]]     # Include current coin

    return table[m][N]



print("Example-1: coin_change_unique_ways_DP(coins, N)")
coins = [1, 2, 5]
print(coin_change_unique_ways_DP(coins, 5))
print("Example-2: coin_change_unique_ways_DP(coins, N)")
coins = [2, 5, 3, 6]
print(coin_change_unique_ways_DP(coins, 10))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 11:20:51 ~/Personal/Notebooks/Algorithms/5. Dynamic Programming $ python3 4_coin_change_unique_ways.py
Example-1: coin_change_unique_ways_DP(coins, N)
4
Example-2: coin_change_unique_ways_DP(coins, N)
5
```

Complexity:

- **Time: O(mn)** :- 2 for loops
- **Auxilliary Space: O(mn)**

5. Coin Change - Min Coins***

Problem:

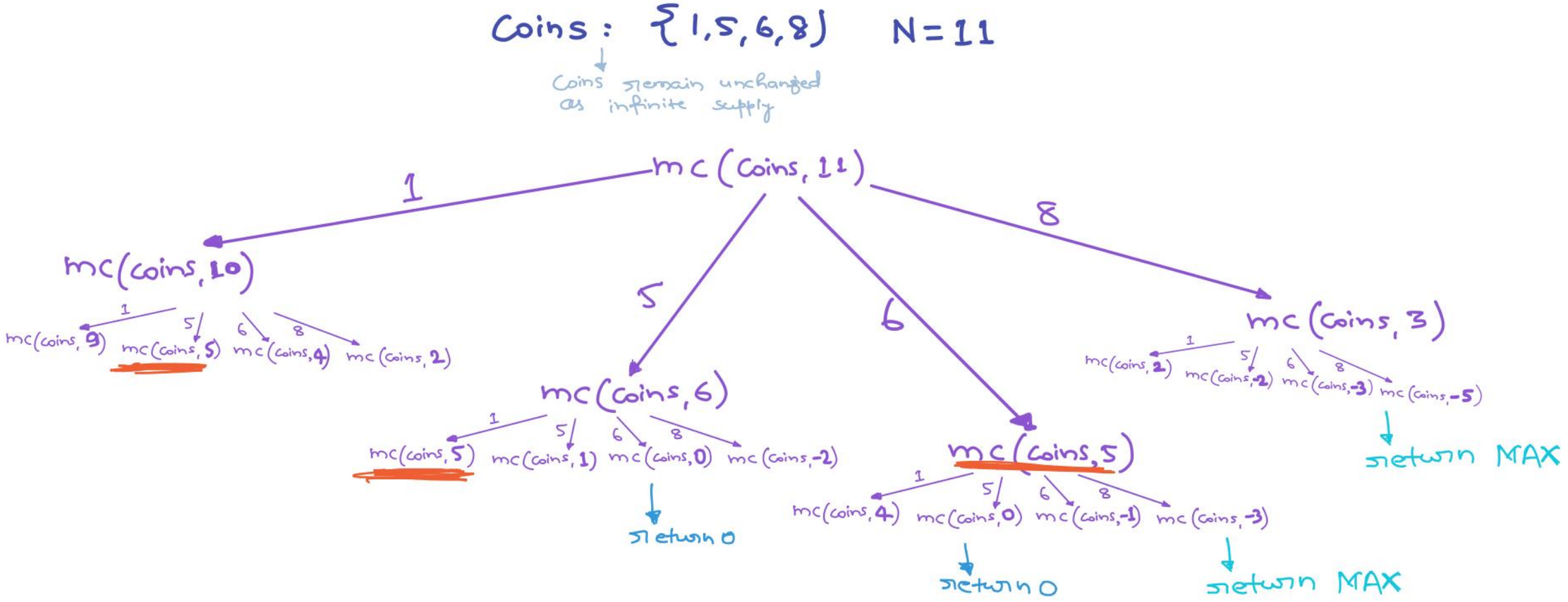
Given a value V, if we want to make change for V cents, and we have infinite supply of each of C = { C1, C2, .. , Cm} valued coins.

What is the minimum number of coins to make the change?

Example:

Input: N = 11 and S = {1, 5, 6, 8} Output: 2 {5, 6}

Recursive Approach:



Implementation

```python
import sys

def coin_change_min_coins_recursive(coins, N):
    m = len(coins)

    # if N==0, No coins needed
    if(N == 0):
        return 0

    # if N < 0, return MAX and also when no coins
    if(N < 0 or m == 0):
        return sys.maxsize

    # Now find minimum no. of coins by including every coin one by one
    min_coins = sys.maxsize
    for i in range(m):
        min_coins = min(1 + coin_change_min_coins_recursive(coins, N-coins[i]), min_coins)

    return min_coins
```

```
print("Example-1: coin_change_min_coins_recursive(coins, N)")
coins = [1, 5, 6, 8]
print(coin_change_min_coins_recursive(coins, 11))

print("Example-2: coin_change_min_coins_recursive(coins, N)")
coins = [2, 5, 7]
print(coin_change_min_coins_recursive(coins, 3))
```

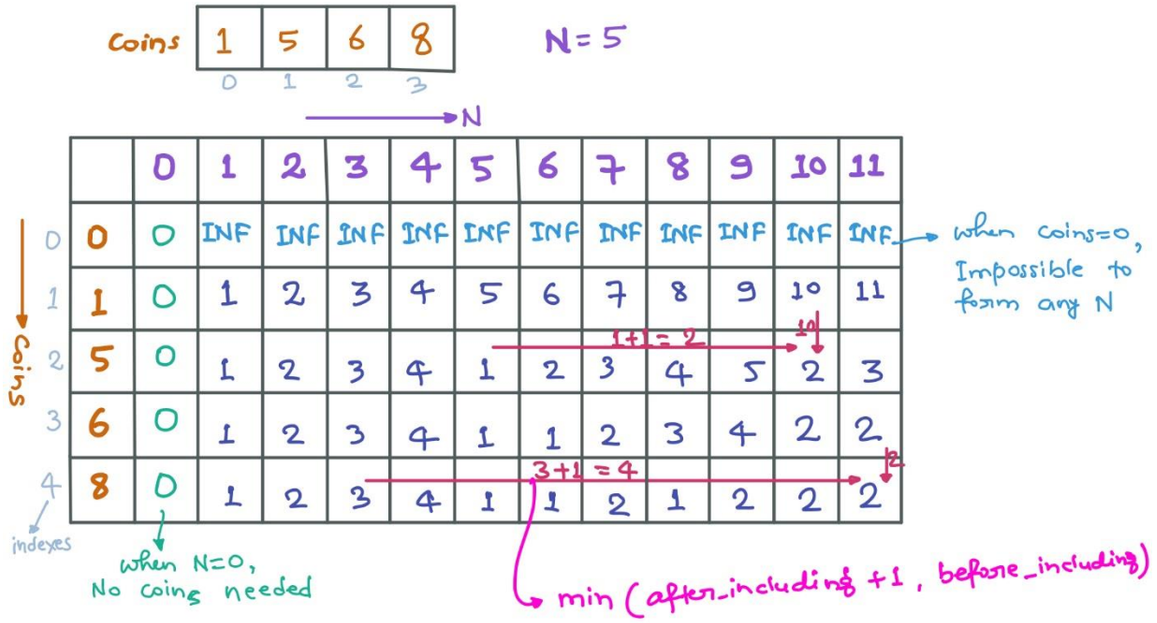
Output:

```
astik.anand@mac-C02XD95ZJG5H 11:14:22 ~/Personal/Notebooks/Algorithms/5. Dynamic Programming $ python3 5_coins_change_min_coins.py
Example-1: coin_change_min_coins_recursive(coins, N)
2
Example-2: coin_change_min_coins_recursive(coins, N)
9223372036854775807
```

Complexity:

- **Time: Exponential**
- **Auxilliary Space: O(n)**

Dynamic Programming Approach:



	0	1	2	3	4	5	6	7	8	9	10	11
0	0	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF
1	0	1	2	3	4	5	6	7	8	9	10	11
2	0	1	2	3	4	1	2	3	4	5	2	3
3	0	1	2	3	4	1	1	2	3	4	2	2
4	0	1	2	3	4	1	1	2	1	2	2	2

Coins: 1 (0), 5 (1), 6 (2), 8 (3)

$$Table[i][j] = min\left(1 + Table[i][j - coins[i-1]],\ Table[i-1, j]\right)$$

Dynamic Programming Implementation:

```
def coin_change_min_coins_DP(coins, N):
    m = len(coins)
    table = [[sys.maxsize]*(N+1) for i in range(m+1)]

    # If N = 0 then coins required = 0 --- Fill first column
    for i in range(m+1):
        table[i][0] = 0

    # If coins = 0 then coins required = INFINITY   --- Fill first column
    # This line can be omitted, just for understanding as we have initialized with infinity
    for j in range(1, N+1):
        table[0][j] = sys.maxsize

    # Now fill the table
    for i in range(1, m+1):
        for j in range(1, N+1):
            if(j-coins[i-1] < 0):
                table[i][j] = table[i-1][j]
            else:
                table[i][j] = min(1+table[i][j-coins[i-1]], table[i-1][j])


    return table[m][N]
```

```
print("Example-1: coin_change_min_coins_DP(coins, N)")
coins = [1, 5, 6, 8]
print(coin_change_min_coins_DP(coins, 11))
print("Example-2: coin_change_min_coins_DP(coins, N)")
coins = [2, 5, 7]
print(coin_change_min_coins_DP(coins, 3))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 13:34:19 ~/Personal/Notebooks/Algorithms/5. Dynamic Programming $ python3 5_coins_change_min_coins.py
Example-1: coin_change_min_coins_DP(coins, N)
2
Example-2: coin_change_min_coins_DP(coins, N)
9223372036854775807
```

Complexity:

- **Time: O(mn)** :- 2 for loops
- **Auxilliary Space: O(mn)**

6. Ways to reach n^{th} stair

Problem:

There are n stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs at a time.

Count the number of ways, the person can reach the top.

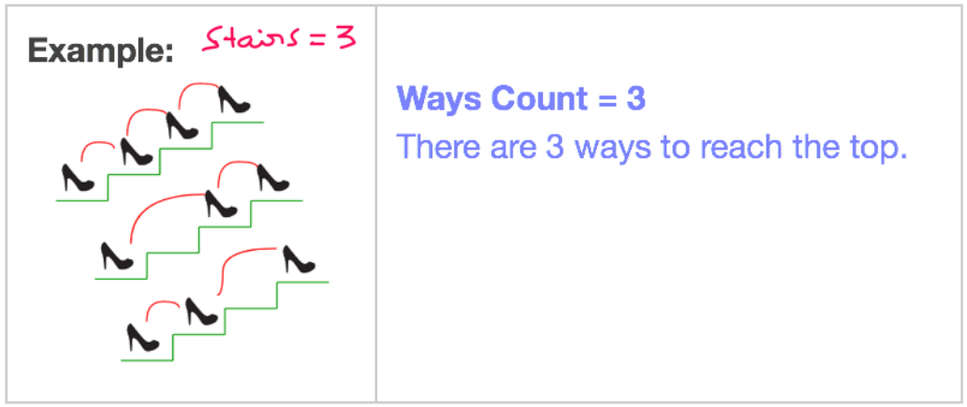
Examples:

Input: n = 1 Output: 1 There is only one way to climb 1 stair

Input: n = 2 Output: 2 There are two ways: (1, 1) and (2)

Input: n = 3 Output: 3 (1, 1, 1), (2, 1), (1, 2)

Input: n = 4 Output: 5 (1, 1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 2)



Recursive Approach:

- If the **total_ways(N)** is the total count of ways to reach Nth stair.
    - If 1 step taken problem reduces to **total_ways(N-1)** .
    - If 2 step taken problem reduces to **total_ways(N-2)** .

    total_ways(N) = total_ways(N-1) + total_ways(N-2)
    total_ways(N) = 1 step taken + 2 steps taken

- Problem can be solved using **fibonacci problem** strategy.

Recursive Implementation:

```
def count_ways_to_nth_stair(N):
    # If N <= 0 :- 0 or less stairs no ways to reach Nth stair
    if(N <= 0):
        return 0

    # If N == 1  :- only 1 way walk 1 stair
    # If N == 2  :- 2 ways ({1,1}, {2})
    if (N == 1 or N == 2):
        return N

    # Else return no. of ways of reaching Nth stair by taking 1 step and 2 steps at once
    return count_ways_to_nth_stair(N-1) + count_ways_to_nth_stair(N-2)



print("Example-1: count_ways_to_nth_stair(3)")
print(count_ways_to_nth_stair(3))
print("Example-2: count_ways_to_nth_stair(4)")
print(count_ways_to_nth_stair(4))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 13:34:20 ~/Personal/Notebooks/Algorithms/5. Dynamic Programming $ python3 6_count_ways_to_nth_stair.py
Example-1: count_ways_to_nth_stair(3)
3
Example-2: count_ways_to_nth_stair(4)
5
```

Dynamic Programming Approach:

If max 2 stairs can be climbed at once, then this problem can be solved by FIBONACCI problem strategy.

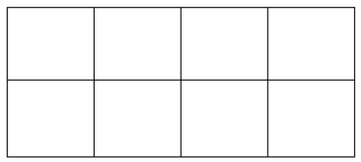
If max m stairs can be climbed at once, then this problem can be solved by MIN_COINS problem strategy.

7. Tiling Problem

Problem:

Given a **"2 x n" board** and **"2 x 1" tiles**, count the **number of ways to tile** the given board using the 2 x 1 tiles.

A tile can either be placed horizontally i.e., as a 1 x 2 tile or vertically i.e., as 2 x 1 tile.



Board

Tile

Example:

Input n = 3 ***Output:*** 3

Explanation:

We need 3 tiles to tile the board of size 2 x 3.

We can tile the board using following ways

1) Place all 3 tiles vertically.

2) Place first tile vertically and remaining 2 tiles horizontally.

3) Place first 2 tiles horizontally and remaining tiles vertically

Input n = 4 ***Output***: 5

Explanation:

For a 2 x 4 board, there are 5 ways

1) All 4 vertical

2) All 4 horizontal

3) First 2 vertical, remaining 2 horizontal

4) First 2 horizontal, remaining 2 vertical

5) Corner 2 vertical, middle 2 horizontal

Recursive Approach:

- Let **total_ways(N)** be the count of ways to place tiles on a *"2 x n" grid*, we have following two ways to place first tile.
    - If we place first tile vertically, the problem reduces to **total_ways(N-1)**.
    - If we place first tile horizontally, we have to place second tile also horizontally and the problem reduces to **total_ways(N-2)**.

total_ways(N) = total_ways(N-1) + total_ways(N-2)

total_ways(N) = 1 step taken + 2 steps taken

This problem like the previous can again be solved using **fibonacci problem** strategy.

Recursive Implementation:

```
def count_tiling_ways(N):
    # If N <= 0 No way to put any tile
    if(N <= 0):
        return 0

    # If N == 1  :- only 1 way put horizontal
    # If N == 2  :- 2 ways either both horizontal or both vertical
    if (N == 1 or N == 2):
        return N

    # Else return no. of ways of filling board by first putting 1 tile vertically or 2 tile horizontally
    return count_tiling_ways(N-1) + count_tiling_ways(N-2)



print("Example-1: count_tiling_ways(3)")
print(count_tiling_ways(3))
print("Example-2: count_tiling_ways(4)")
print(count_tiling_ways(4))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 21:40:57 ~/Personal/Notebooks/Algorithms/5. Dynamic Programming $ python3 7_count_ways_of_tiling.py
Example-1: count_tiling_ways(3)
3
Example-2: count_tiling_ways(4)
5
```

8. Possible Decodings of Digit Sequence***

Problem:

Let 1 represent 'A', 2 represents 'B', . . . 26 represents 'Z' etc.

Given a digit sequence, count the number of possible decodings of the given digit sequence.

Examples:

Input: 121 Output: 3 coz Possible Decodings are: "ABA", "AU", "LA"

Input: 1234 Output: 3 coz Possible Decodings are: "ABCD", "LCD", "AWD"

Recursive Approach

- This problem is recursive and can be broken in sub-problems.
- Start from end of the given digit sequence.
- Initialize the total count of decodings as 0.
- Recur for two subproblems.
    i. If the last digit is non-zero, recur for remaining (n-1) digits and add the result to total count.
    ii. If the last two digits form a valid character (or smaller than 27), recur for remaining (n-2) digits and add the result to total count.

Implementation:

```
def count_digits_sequence_decodings(digits):
    n = len(digits)

    # base cases
    if(n==0 or n==1):
```

```
        return 1

    count = 0

    # If the last digit is not 0, then last digit must add to the number of words.
    if(digits[-1]>'0'):
        count = count_digits_sequence_decodings(digits[:-1])

    # If the last two digits form a number smaller than or equal to 26, then consider last two digits and recur
    if(digits[-2]=='1' or (digits[-2]=='2' and digits[-1]<='6')):
        count += count_digits_sequence_decodings(digits[:-2])

    return count


print("Recursive Approach")
print("Example-1: count_digits_sequence_decodings('12')")
print(count_digits_sequence_decodings('12'))

print("Example-1: count_digits_sequence_decodings('121')")
print(count_digits_sequence_decodings('121'))

print("Example-3: count_digits_sequence_decodings('1234')")
print(count_digits_sequence_decodings('1234'))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 10:45:38 ~/Personal/Notebooks/Algorithms/9. Mathematical Algorithms $ python3 8_digits_sequence_decodings.py
Example-1: count_digits_sequence_decodings('12')
2
Example-1: count_digits_sequence_decodings('121')
3
Example-3: count_digits_sequence_decodings('1234')
3
```

Complexity:

- **Time: Exponential**

Dynamic Programming Approach

- If we take a closer look at the above program, we can observe that the recursive solution is similar to Fibonacci Numbers.
- Therefore, we can optimize the above solution to work in **O(n)** time using Dynamic Programming.

DP Implementation

```python
def count_digits_sequence_decodings_DP(digits):
    n = len(digits)
    count = [0]*(n+1) # A table to store results of subproblems

    # Base cases
    count[0] = 1;  count[1] = 1

    for i in range(2, n+1):
        count[i] = 0

        # If the last digit is not 0, then last digit must add to the number of words
        if (digits[i-1] > '0'):
            count[i] = count[i-1]

        # If the last two digits form a number smaller than or equal to 26,
        # then last two digits form a valid character
        if (digits[i-2]=='1' or (digits[i-2]=='2' and digits[i-1]<='6') ):
            count[i] += count[i-2]

    return count[n]


print("\nDP Approach")
print("Example-1: count_digits_sequence_decodings_DP('12')")
print(count_digits_sequence_decodings_DP('12'))

print("Example-1: count_digits_sequence_decodings_DP('121')")
print(count_digits_sequence_decodings_DP('121'))

print("Example-3: count_digits_sequence_decodings_DP('1234')")
print(count_digits_sequence_decodings_DP('1234'))

print("Example-3: count_digits_sequence_decodings_DP('1234121')")
print(count_digits_sequence_decodings_DP('1234121'))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 11:06:40 ~/Personal/Notebooks/Algorithms/9. Mathematical Algorithms $ python3 8_digits_sequence_decodings.py
Example-1: count_digits_sequence_decodings_DP('12')
2
Example-1: count_digits_sequence_decodings_DP('121')
3
Example-3: count_digits_sequence_decodings_DP('1234')
3
```

Complexity:

- **Time: O(n)**

Divide and Conquer Approach

Like Greedy and Dynamic Programming, Divide and Conquer is an algorithmic paradigm.
A typical Divide and Conquer algorithm solves a problem using following three steps.

- ***Divide:*** Break the given problem into subproblems of same type.
- ***Conquer:*** Recursively solve these subproblems
- ***Combine:*** Appropriately combine the answers

Standard Divide and Conquer Algorithms

- **Binary Search:**
  - A searching algorithm.
  - In each step, the algorithm compares the input element x with the value of the middle element in array.
  - If the values match, return the index of middle.
  - Otherwise, if x is less than the middle element, then the algorithm recurs for left side of middle element, else recurse for right side of middle element.
- **Quicksort:**
  - A sorting algorithm.
  - The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side.
  - Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.
- **Mergesort:**
  - Also a sorting algorithm.
  - The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.
- **Closest Pair of Points:**
  - The problem is to find the closest pair of points in a set of points in x-y plane.
  - Problem can be solved in $O(n^2)$ time by calculating distances of every pair of points & comparing the distances to find the minimum.
  - Divide and Conquer algorithm solves the problem in O(nLogn) time.
- **Strassen's Algorithm:**
  - An efficient algorithm to multiply two matrices.
  - A simple method to multiply two matrices need 3 nested loops and is $O(n^3)$.
  - Strassen's algorithm multiplies two matrices in $O(n^{2.8974})$ time.
- **Cooley-Turkey Fast Fourier transform(FFT) Algorithm:**
  - The most common algorithm for FFT.
  - It is a divide and conquer algorithm which works in O(nlogn) time.

- **Kartsuba Algorithm for fast multiplication:**
  - It does multiplication of two *n*-digit numbers in at most $\mathbf{3n^{\log_2 3}} \approx \mathbf{3n^{1.585}}$ single-digit multiplications in general (and exactly $\mathbf{3n^{\log_2 3}}$ when n is a power of 2.
  - If $n = 2^{10} = 1024$, in particular, the exact counts are $3^{10} = 59,049$ and $(2^{10})^2 = 1,048,576$, respectively.

Divide and Conquer Vs. Dynamic Programming

- Both paradigms (D & C and DP) divide the given problem into subproblems and solve subproblems.
- **How to choose one of them for a given problem?**
  - ***Divide and Conquer should be used when same subproblems are not evaluated many times.***
  - ***Otherwise Dynamic Programming or Memoization should be used.***
  - For example: Binary Search is a Divide and Conquer algorithm, we never evaluate the same subproblems again.
  - On the other hand, for calculating nth Fibonacci number, Dynamic Programming should be preferred.

Standard Divide and Conquer Problems

1. Calculate Power(x, n)

Problem:

Given two integers k and n, write a function to compute kn. We may assume that k and n are small and overflow doesn't happen.

Examples:

Input: k = 2, n = 3 Output: 8

Input: k = 7, n = 9 Output: 40353607

Approach:

- Idea is to divide the problem into halves. Like $\mathbf{k^n = k^{n/2} \times k^{n/2}}$
- If n is odd: $\mathbf{k^n = k \times k^{n/2} \times k^{n/2}}$
- If n is even: $\mathbf{k^n = k^{n/2} \times k^{n/2}}$

Implementation:

```
def power(k, n):
    if(n==0):
        return 1
```

```
    temp = power(k, n//2)
    if(n%2==0):
        return temp*temp
    else:
        return k*temp*temp



print("Example-1: power(2, 3)")
print(power(2, 3))

print("Example-2: power(7, 9)")
print(power(7, 9))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 02:46:41 ~/Personal/Notebooks/Algorithms/6. Divide & Conquer Algorithms $ python3 1_power_k_n.py
Example-1: power(2, 3)
8
Example-2: power(7, 9)
40353607
```

Complexity:

- **Time: O(logn)**
- **Auxilliary Space: O(1)**

2. Median of 2 Sorted Arrays of Same Size***

Problem:

There are 2 sorted arrays A and B of size n each.Write an algorithm to find the median of the array obtained after merging the above 2 arrays(i.e. array of length 2n).

The complexity should be O(log(n)).

Example:

Input: arr1 = [1, 12, 15, 26, 38] arr2 = [2, 13, 17, 30, 45]

Output: 16

Explanation:

After merging: arr = [1, 2, 12, 13, 15, 17, 26, 30, 38, 45]

Middle 2 elements are 15 and 17 and hence (15+17)/2 = 16

Approach-1: Simple

Count while merging and once count reaches n coz total 2n elements get the median.

Time Complexity: O(n)

Approach-2: Divide and Conquer

- Calculate the medians m1 and m2 of the input arrays arr1[ ]and arr2[ ] respectively.
- If m1 and m2 both are equal then we are done. return m1 (or m2)
- If m1 is greater than m2, then median is present in one of the below two subarrays.
a) From first element of arr1 to m1 (arr1[0…	*n/2*	])
b) From m2 to last element of arr2 (arr2[	*n/2*	…n-1])
- If m2 is greater than m1, then median is present in one of the below two subarrays.
a) From m1 to last element of arr1 (arr1[	*n/2*	…n-1])
b) From first element of arr2 to m2 (arr2[0…	*n/2*	])
- Repeat the above process until size of both the subarrays becomes 2.
- If size of the two arrays is 2 then use formula to get the median.

  Median = (max(arr1[0], arr2[0]) + min(arr1[1], arr2[1]))/2

Implementation:

```
def median(arr):
    n = len(arr)
    if(n%2==0):
        return (arr[(n//2)-1] + arr[n//2])/2
```

```
    else:
        return arr[n//2]


def find_median(arr1, arr2):
    n = len(arr1)

    if(n==0):
        return -1
    # 1 element in each: Median = (arr1[0] + arr2[0])/2
    elif(n==1):
        return (arr1[0] + arr2[0])/2
    # 2 elements in each: Median = (max(arr1[0], arr2[0]) + min(arr1[1], arr2[1]))/2
    elif(n==2):
        return (max(arr1[0], arr2[0]) + min(arr1[1], arr2[1]))/2
    else:
        # Median of the 2 arrays for comparison
        M1 = median(arr1)
        M2 = median(arr2)

        # We are done return median
        if(M1 == M2):
            return M1
        # As M1>M2 so, median should lie b/w arr1[.....M1] and arr2[M2......]
        elif(M1>M2):
            if(n%2==0):
                return find_median(arr1[:n//2], arr2[n//2:])
            else:
                return find_median(arr1[:n//2+1], arr2[n//2:])
        # M1<M2 so, median should lie b/w arr1[M1.....] and arr2[.....M2]
        else:
            if(n%2==0):
                return find_median(arr1[n//2:], arr2[:n//2])
            else:
                return find_median(arr1[n//2:], arr2[:n//2+1])



print("Example-1: find_median(arr1, arr2)")
arr1 = [1, 2, 3, 6]
arr2 = [4, 6, 8, 10]
print(find_median(arr1, arr2))

print("Example-2: find_median(arr1, arr2)")
arr1 = [1, 12, 15, 26, 38]
arr2 = [2, 13, 17, 30, 45]
print(find_median(arr1, arr2))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 08:11:51 ~/Personal/Notebooks/Algorithms/6. Divide & Conquer Algorithms $ python3 2_find_median.py
Example-1: find_median(arr1, arr2)
5.0
Example-2: find_median(arr1, arr2)
16.0
```

Complexity:

- **Time: O(logn)**
- **Auxilliary Space: O(1)**

3. Count Inversions in an Array***

What is inversion count?

Inversion Count for an array indicates – how far (or close) the array is from being sorted.

If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.

Formally, if two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j.

Approach-1: Simple

For each element just count the number of smaller elements than it and are present on right side

Time Complexity: $O(n^2)$

Approach-2: Divide and Conquer (Enhance Merge Sort Approach)

Let the inversions count of **left half** of array be $\mathbf{INV_1}$ and **right half** of the array $\mathbf{INV_2}$.

- What kinds of inversions are still missing for in $\mathbf{INV_1 + INV_2}$?
- The answer is – the inversions we have to count during the merge step $\mathbf{INV_{merge}}$.

Total inversions = $INV_1 + INV_2 + INV_{merge}$

Implementation:

```
def count_inv(arr):
    n = len(arr)
    # Return back the array with inversion_count=0 if size is 0 or 1
    if(n==0 or n==1):
        return (0, arr)

    # Get the inversion_count of left subarray with merge_sort approach
    INV_1, sorted_arr_1 = count_inv(arr[:n//2])
    # Get the inversion_count of right subarray with merge_sort approach
    INV_2, sorted_arr_2 = count_inv(arr[n//2:])
    # Get the inversion_count for merging
```

```
    INV_MERGE, sorted_arr = count_inv_merge(sorted_arr_1, sorted_arr_2)

    return (INV_1+INV_2+INV_MERGE, sorted_arr)


def count_inv_merge(arr1, arr2):
    n1 = len(arr1)
    n2 = len(arr2)
    count = 0
    result = []
    i=0; j=0
    while(i<n1 and j<n2):
        if(arr1[i] <= arr2[j]):
            result.append(arr1[i])
            i+=1
        else:
            result.append(arr2[j])
            # Jab bhi right subarray(arr2) ka koi element pick hota hai that means
            # It is smaller than all the elements to the right of i till n1 of left subarray(arr1)
            # and wo in saare elements ke liye inversions dega. So  count += n1-i
            count+= n1-i
            j+=1

    if(i<n1):
        result += arr1[i:]

    if(j<n2):
        result += arr2[j:]

    return (count, result)



print("Example-1: count_inv(arr)" )
arr = [2, 4, 1, 3, 5]
print(count_inv(arr)[0])


print("Example-2: count_inv(arr)" )
arr = [1, 20, 6, 4, 5]
print(count_inv(arr)[0])
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 10:29:56 ~/Personal/Notebooks/Algorithms/6. Divide & Conquer Algorithms $ python3 3_count_inversions.py
Example-1: count_inv(arr)
3
Example-2: count_inv(arr)
5
```

Complexity:

- **Time: O(nlogn)**

- **Auxilliary Space: O(n)**

Problems To Do:

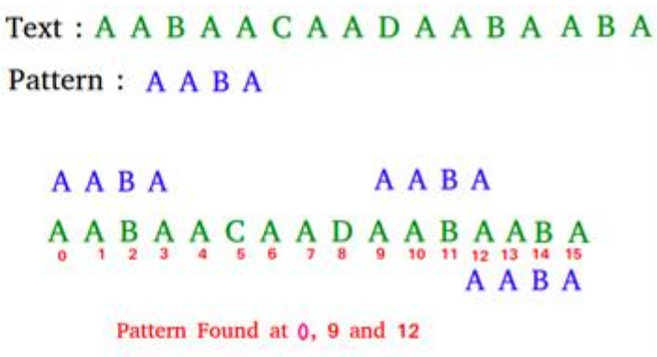
- Strassen's Matrix Multiplication

Pattern Search Algorithms

Why Pattern Search Algorithms ?

- Many a times we need to find a pattern **pat[0..m-1]** in a given text **txt[0..n-1]** assuming n >= m.
- In those situation we use pattern search algorithms.
- **Example:**

```
Input:  txt[] = "THIS IS A TEST TEXT"
        pat[] = "TEST"
Output: Pattern found at index 10

Input:  txt[] =  "AABAACAADAABAABA"
        pat[] =  "AABA"
Output: Pattern found at index 0
        Pattern found at index 9
        Pattern found at index 12
```



- Pattern searching is an important problem in computer science.
- When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

Famous Pattern Search Algorithms

- Naive Algorithm

- Rabin-Karp Algorithm
- KMP (Knuth-Morris-Pratt) Algorithm

Algo-1: Naive Algorithm

- Just need to use 2 loops to check every substring of same length as of given pattern and check if it matches it.

Implementation:

```
# Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(pat, txt) that prints all occurrences of pat[] in txt[].
# You may assume that n > m.


def naive_pattern_search(pat, text):
    m = len(pat)
    n = len(text)

    for i in range(n-m+1):
        flag = True
        for j in range(m):
            if(pat[j] != text[i+j]):
                flag = False

        if(flag):
            print("Pattern found at index {}".format(i))


print("Example-1:")
txt = "AABAACAADAABAABA"
pat = "AABA"
naive_pattern_search(pat, txt)

print("Example-2:")
txt = "abracadabra"
pat = "ab"
naive_pattern_search(pat, txt)


# Time Complexity:
# Best Case: O(n) - First character of pattern is not present in text
# Worst Case: O(n-m+1)(m) - All characters of the text and pattern are same OR only the last character is different.
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 20:52:48 ~/Personal/Notebooks/Algorithms/7. String Algorithms  $ python3 1_naive_algo.py
Example-1:
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12

Example-2:
Pattern found at index 0
Pattern found at index 7
```

Complexity:

- **Time:**
  - ***Best Case:*** **O(n)** - First character of pattern is not present in text
  - ***Worst Case***: **O(n-m+1)(m) -** All characters of the text and pattern are same OR only the last character is different.
- **Auxilliary Space: O(1)**

Algo-2: Rabin-Karp Algorithm***

- Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one.
- But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text,and if the hash values match then only it starts matching individual characters.
- **Rabin-Karp algorithm needs to calculate hash values for following strings:**
  i. Pattern itself.
  ii. All the substrings of text of length m.
- **Calculating hash function as suggested by Rabin-Karp:**
  - We treat string character with its ASCII value and hence we take base as 256.
  - The numeric values cannot be practically stored as an integer.
  - Hence numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable.
  - To do rehashing, we need to take off the most significant digit and add the new least significant digit for hash value.

Rehashing Formula:

hash(txt[s+1 .. s+m]) = (d (hash(txt[s .. s+m-1]) – txt[s]*h) + txt[s + m]) mod q

Implementation:

```
def rabin_karp_search(pat, text):
    base = 256          # Total number of characters, if you consider less numbers of characters
                        # base can be set accordingly
    q = 101             # Prime to take modulo
    p_hash = 0          # Hash value for pattern
    t_hash = 0          # Hash value for text
    m = len(pat)
    n = len(text)

    # Calculating h that will be used for rehashing:  h = pow(base, m-1)%q
    h = 1
    for i in range(m-1):
        h = (h*base)%q

    # Calculating hash for pat and initial text
    for i in range(m):
        p_hash = (p_hash*base + ord(pat[i]))%q      # ord(char) : gives ascii value of char
        t_hash = (t_hash*base + ord(text[i]))%q

    # Slide the pattern over text one by one
    for i in range(n-m+1):
        # Check the hash values of current window of text and pattern
        if t_hash == p_hash:   # if the hash values match then only check for characters one by one
            for j in range(m):
                if(text[i+j] != pat[j]):
                    break
            if(j==m-1):
                print("Found at index {}".format(i))

        # Calculate hash value for next window of text: Remove leading digit, add trailing digit
        if i+m < n:
            t_hash = ((t_hash - h*ord(text[i]))*base + ord(text[i+m]))%q

            # We might get negative values of t_hash, converting it to positive
            if t_hash < 0:
                t_hash = t_hash+q


print("Rabin-Karp Example-1:")
txt = "bacbababababacaca"
pat = "ababaca"
rabin_karp_search(pat, txt)

print("\nRabin-Karp Example-2:")
txt = "abracadabra"
pat = "ab"
rabin_karp_search(pat, txt)

print("\nRabin-Karp Example-3:")
txt = "AABAACAADAABAABA"
pat = "AABA"
rabin_karp_search(pat, txt)

# Complexity:
# Time: O(n)
```

```
# Auxilliary Space: O(1)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 17:20:34 ~/Personal/Notebooks/Algorithms/7. String Algorithms  $ python3 2_rabin_karp_algo.py
Rabin-Karp Example-1:
Found at index 6

Rabin-Karp Example-2:
Found at index 0
Found at index 7

Rabin-Karp Example-3:
Found at index 0
Found at index 9
Found at index 12
```

Complexity:

- **Time:**
  - ***Average & Best Case:*** **O(n + m)**
  - ***Worst Case***: **O(n)(m)** - Failure of hash function that leads to all spurious hits
- **Auxilliary Space: O(1)**

Algo-3: KMP (Knuth Morris Pratt) Algorithm***

- Naive doesn't work well in cases where we see many matching characters followed by a mismatching character.
- ***Example:*** **txt** = "AAAAAAAAAAAAAAAAB" and **pat** = "AAAAB"
- KMP algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern & improves the worst case complexity to O(n).
- The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window.
- We take advantage of this information to avoid matching the characters that we know will anyway match.

Step-1: Preprocessing

- KMP algorithm preprocesses **pat[]** and constructs an auxiliary **lps[]** of size m (same as size of pattern) which is used to skip characters while matching.
- Name lps indicates longest proper prefix which is also suffix.

- A proper prefix is prefix with whole string notallowed.

  Example:
  Prefixes of “ABC” are “ “, “A”, “AB” and “ABC”.
  Proper prefixes are “ “, “A” and “AB”.
  Suffixes of the string are “ “, “C”, “BC” and “ABC”.

- We search for lps in sub-patterns. More clearly we focus on sub-strings of patterns that are either prefix and suffix.
- For each sub-pattern **pat[0..i]** where i = 0 to m-1, **lps[i]** stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern **pat[0..i]**.
- For each sub-pattern **pat[0..i]** where i = 0 to m-1, **lps[i]** stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern **pat[0..i]**.

  lps[i] = the longest proper prefix of pat[0..i] which is also a suffix of pat[0..i]

Longest Proper Prefix also a Suffix: LPS[]

- Calculating LPS or Pi (π) or False function

	0	1	2	3	4	5	6
Pat	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

	0	1	2	3	4	5	6	7	8	9
Pat	a	b	c	d	a	b	e	a	b	f
π	0	0	0	0	1	2	0	1	2	0

Algorithm to calculate π

- Start from **j=0, i=1** and **π[0]=0** for pat.
- if **pat[i]==pat[j]** put **π[i]=j+1** and **increase** both **i** and **j**.
- Else if it doesn’t match and **j>0** change **j = π[j-1]**.
- Else if j==0 put **π[i]=0** and **increase i**.

Implementation to calculate π

```
def calculate_lps(pat):
    m = len(pat)
    lps = [0]*m
    i = 1; j = 0

    while(i < m):
        if(pat[i] == pat[j]):
            lps[i] = j+1
            i+=1
            j+=1
        elif(j>0):
            j = lps[j-1]
        else:
            lps[i] = 0
```

```
            i+=1

    return lps


print("LPS Example-1:")
pat = "abababaca"
print(str(calculate_lps(pat)))

print("\nLPS Example-2:")
pat = "abcdabeabf"
print(str(calculate_lps(pat)))

# Complexity:
# Time: O(m)
# Auxilliary Space: O(m)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 05:22:20 ~/Personal/Notebooks/Algorithms/7. String Algorithms  $ python3 2_kmp_algo.py
LPS Example-1:
[0, 0, 1, 2, 3, 0, 1]

LPS Example-2:
[0, 0, 0, 0, 1, 2, 0, 1, 2, 0]
```

Complexity:

- **Time: O(m)**
- **Auxilliary Space: O(m)**

Step-2: Matching Algorithm

- Start from **i=0, j=0** we will use i to track text and j to track pat.
- if **text[i] == pat[j]** then **increase** both **i** and **j** and check
    - if j==m then it's match print and reset **j = lps[j-1]** to find next occurrence of the pattern
- Else if it doesn't match and **j>0** change **j = lps[j-1]**.
- Else if j==0 then **increase i**.

Implementation

```
def KMP_search(pat, text):
    lps = calculate_lps(pat)
    i=0; j=0
    m = len(pat)
    n = len(text)
```

```
    while(i<n):
        if(text[i] == pat[j]):
            i+=1
            j+=1
            if(j == m):
                print("Found at index {}".format(i-j))
                j = lps[j-1]
        elif(j>0):
            j = lps[j-1]
        else:
            i+=1

print("KMP Example-1:")
txt = "bacbababababacaca"
pat = "ababaca"
KMP_search(pat, txt)

print("\nKMP Example-2:")
txt = "abracadabra"
pat = "ab"
KMP_search(pat, txt)

print("\nKMP Example-3:")
txt = "AABAACAADAABAABA"
pat = "AABA"
KMP_search(pat, txt)

# Complexity:
# Time: O(m+n)
# Auxilliary Space: O(m)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 17:21:08 ~/Personal/Notebooks/Algorithms/7. String Algorithms  $ python3 3_kmp_algo.py
KMP Example-1:
Found at index 6

KMP Example-2:
Found at index 0
Found at index 7

KMP Example-3:
Found at index 0
Found at index 9
Found at index 12
```

Complexity:

- **Time: O(n+m)**
- **Auxilliary Space: O(m)**

Standard Pattern Search Algorithms Problmes

1. Anagram Substring Search

Problem:

Given a text **txt[0..n-1]** and a pattern **pat[0..m-1]**.

Write a function **search(pat, txt)** that prints **all occurrences of pat[] and its permutations (or anagrams)** in txt[].

You may assume that n > m.

```
1) Input:  txt[] = "BACDGABCDA"  pat[] = "ABCD"
   Output:   Found at Index 0
             Found at Index 5
             Found at Index 6
2) Input: txt[] =  "AAABABAA" pat[] = "AABA"
   Output:   Found at Index 0
             Found at Index 1
             Found at Index 4
```

Algorithm:

- Store counts of frequencies of pattern in first count array **countP[]** and counts of frequencies of characters in first window of text in array **countTW[]**.
- Now run a loop from i =m to n-1. Do following in loop.
    - a) If the two count arrays are identical, we found an occurrence.
    - b) Increment count of current character of text in countTW[]
    - c) Decrement count of first character in previous window in countWT[]
- The last window is not checked by above loop, so explicitly check it.

Implementation

```
def anagram_search(pat, text):
    m = len(pat)
    n = len(text)
    count_pat = [0]*256     # Total 256 characters
    count_text_window = [0]*256

    # fill the count_pat and count_text_window array
    for i in range(m):
        count_pat[ord(pat[i])] += 1
        count_text_window[ord(pat[i])] += 1

    # Starts searching anagrams or permutations
    for i in range(m, n):
        if(count_pat == count_text_window):
            print("Found at index {}".format(i-m))

        # Increase the frequency of next character and decrease the frequncy of first character in window
        count_text_window[ord(text[i])] += 1
        count_text_window[ord(text[i-m])] -= 1

    # Check for last window as it will be left in this loop
    if(count_pat == count_text_window):
            print("Found at index {}".format(n-m))


print("Anagram Search Example-1:")
txt = "BACDGABCDA"
pat = "ABCD"
anagram_search(pat, txt)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 20:38:50 ~/Personal/Notebooks/Algorithms/7. String Algorithms  $ python3 4_anagram_search.py
Anagram Search Example-1:
Found at index 0
Found at index 5
Found at index 6
```

Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(1) :** Only 256 spaces can be treated as constant.

2. Manacher's Algorithm - O(N) Time Longest Palindromic Substring***

Problem:

Given a string, find the longest substring which is palindrome.

Examples:

- string = "abaabc" output = "baab"
- string = "babcbabcbaccba" output = "abcbabcba"
- string = "abaaba" output = "abaaba"
- string = "abababa" output = "abababa"
- string = "forgeeksskeegfor" output = "geeksskeeg"

Approaches to solve:

- **Naive Approach :** $O(n^3)$
- **Dynamic Programming:** $O(n^2)$
- **Manacher Algorithm:** O(n)

Manacher Approach:

string = "abaabc" new_text = "#a#b#a#a#b#c#"

#	a	#	b	#	a	#	a	#	b	#	c	#
0	1	0	3	0	1	4	1	0	1	0	1	0

Manacher Algorithm:

- **Pre-Processing:**
  - Push **'#'** before and after every character and create a **new_text**. Example: text = "abc" then new_text= "#a#b#a#"
  - Create an array **P[]** of size(2n+1) to store palindromic values of the new_text.
  - Initialize **Center(C) = 0** and **Right(R) = 0**
- **Processing:**
  - Loop over the pre-processed new_text and do
  - Get **mirror_index** of currently processing value(i) using center(C).
  - Check if Right(R) is greater than current(i): then put **min(R-i, P[i_mirror])** or else put 0 in P[i].

    - Keep expanding using current and already palindrome (at that stage using p[I]) until the next character and mirror character from i are same.
    - Now if newly calculated P[i] is greater than R-i then update **C=i** and **R=P[i] + i**
- **Post-Processing:**
    - Find the max of P[] array and that is the max_length of palindrome and its index is center_index
    - Calculate start and end index of palindrome in original text using start = (center_index-max_pal_len)/2 and end = (center_index+max_pal_len)/2
    - Print the palindrome using start and end index using original text.

Implementation

```
def manacher_longest_palindromic_substring(text):
    # Preprocess the given text
    new_text = "#"
    for ch in text:
        new_text += ch
        new_text += "#"

    n = len(new_text)
    P = [0]*n    # Array to store palindromic values

    # Center(C) : Index of mirror line
    # Right(R)  : Where we are going to end our bounds on right hand side
    # Left(L)   : No need to store Left(L) as it can be calculated using something like i_mirror
    C = 0
    R = 0

    # Iterate through entire new_text
    for i in range(n):
        # i is pointing to current element
        mirror = C - (i-C)  # Mirror index of i

        # If i < right bound(R) then simply copy min of mirror value or (R-i)
        if(i < R):
            P[i] = min(R-i, P[mirror])

        # Expand around both sides of i to find new center(C)
        while(((i + P[i] + 1) < n) and ((i - P[i] - 1) >= 0) and \
              (new_s[i + P[i] + 1] == new_s[i - P[i] - 1])):
            P[i] += 1

        # If i + P[i] > right bound(R) then update Center and Right bound
        if(i + P[i] > R):
            C = i
            R = i + P[i]

    # Find the length of largest palindrome and center_index
    max_pal_len = max(P)
    center_index = P.index(max_pal_len)

    # Print the longest palindrome
    start = int((center_index-max_pal_len)/2)
    end = int((center_index+max_pal_len)/2)

    print("Largest Palindromic Susbtring: {}".format(text[start:end]))
```

```
print("Manacher Example-1:")
txt = "abaabc"
manacher_longest_palindromic_substring(txt)

print("Manacher Example-2:")
txt = "babcbabcbaccba"
manacher_longest_palindromic_substring(txt)

print("Manacher Example-3:")
txt = "abaaba"
manacher_longest_palindromic_substring(txt)

print("Manacher Example-4:")
txt = "abababa"
manacher_longest_palindromic_substring(txt)

print("Manacher Example-5:")
txt = "forgeeksskeegfor"
manacher_longest_palindromic_substring(txt)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 09:37:47 ~/Personal/Notebooks/Algorithms/7. String Algorithms  $ python3 5_manacher_longest_palindromic_substring.py
Manacher Example-1:
Largest Palindromic Susbtring: baab
Manacher Example-2:
Largest Palindromic Susbtring: abcbabcba
Manacher Example-3:
Largest Palindromic Susbtring: abaaba
Manacher Example-4:
Largest Palindromic Susbtring: abababa
Manacher Example-5:
Largest Palindromic Susbtring: geeksskeeg
```

Complexity:

- **Time: O(n) :** Traversing through loop only once.
- **Auxilliary Space: O(n) :** We are using (2n+1) spaces array

3. All Possible Strings - Made by Placing Spaces***

Problem:

Given a string, print all possible strings that can be made by placing spaces (zero or one) in between them.

Example:

input = ABC

output:

ABC

AB C

A BC

A B C

Algorithm:

- Use recursion and create a buffer that one by one contains all output strings having spaces.
- Keep updating buffer in every recursive call.
- If the length of given string is **"n"** the updated string can have maximum length of **2n-1** and so we create buffer size of 2n (one extra character for string termination).
- Leave 1st character as it is, starting from the 2nd character, either fill a space or a character.

Implementation:

```
def to_string(str_list):
    s = ""
    for x in str_list:
        if x == "$":
            break
        s += x
    return s


def print_pattern_util(string, buffer, i, j, n):
    if(i==n):
        buffer[j] = "$"
        print(to_string(buffer))
        return

    # Either put the character
    buffer[j] = string[i]
    print_pattern_util(string, buffer, i+1, j+1, n)

    # Or put a space followed by next character
    buffer[j] = " "
    buffer[j+1] = string[i]

    print_pattern_util(string, buffer, i+1, j+2, n)
```

```python
def print_pattern(string):
    n = len(string)
    buffer = [0]*(2*n)
    buffer[0] = string[0]

    print_pattern_util(string, buffer, 1, 1, n)



print("Example-1:")
string = "ABCD"
print_pattern(string)

print("Example-2:")
string = "12345"
print_pattern(string)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 13:02:50 ~/Personal/Notebooks/Algorithms/7. String Algorithms  $ python3 7_all_possible_string_using_spaces.py
Example-1:
ABCD
ABC D
AB CD
AB C D
A BCD
A BC D
A B CD
A B C D
Example-2:
12345
1234 5
123 45
123 4 5
12 345
12 34 5
12 3 45
12 3 4 5
1 2345
1 234 5
1 23 45
1 23 4 5
1 2 345
1 2 34 5
1 2 3 45
1 2 3 4 5
```

Complexity:

- **Time: O(n*(2^n)) :** Since number of Gaps are n-1, there are total 2^{n-1} patterns each having length ranging from n to 2n-1, hence overall complexity would be O(n*(2^n))
- **Auxilliary Space: O(n) :** We are using (2n) spaces array

Problems To Do:

- Longest Even Length Substring :- Sum of 1st & 2nd half is same
- Print all Anagrams Together - Sequence of words are given

Bit Algorithms

Working on bytes, or data types comprising of bytes like ints, floats, doubles or even data structures which stores large amount of bytes is normal for a programmer. In some cases, a programmer needs to go beyond this - that is to say that in a deeper level where the importance of bits is realized.

Operations with bits are used in **Data compression** (data is compressed by converting it from one representation to another, to reduce the space) ,**Exclusive-Or Encryption** (an algorithm to encrypt the data for safety issues). In order to encode, decode or compress files we have to extract the data at bit level. Bitwise Operations are faster and closer to the system and sometimes optimize the program to a good level.

We all know that 1 byte comprises of 8 bits and any integer or character can be represented using bits in computers, which we call its binary form(contains only 1 or 0) or in its base 2 form.

Example: 1) 14 = $\{1110\}_2 = 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 14$.

2) 20 = $\{10100\}_2 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = 20$.

For characters, we use ASCII representation, which are in the form of integers which again can be represented using bits.

Bitwise Operators:

There are different bitwise operations used in the bit manipulation. These bit operations operate on the individual bits of the bit patterns. Bit operations are fast and can be used in optimizing time complexity.

Some common bit operators are:

- **NOT (~):** Bitwise NOT is an unary operator that flips the bits of the number i.e., if the ith bit is 0, it will change it to 1 and vice versa. Bitwise NOT is nothing but simply the one's complement of a number. Lets take an example.

  $N = 5 = (101)_2$ $\sim N = \sim 5 = \sim(101)_2 = (010)_2 = 2$

- **AND (&):** Bitwise AND is a binary operator that operates on two equal-length bit patterns. If both bits in the compared position of the bit patterns are 1, the bit in the resulting bit pattern is 1, otherwise 0.

  $A = 5 = (101)_2$, $B = 3 = (011)_2$

  $A \& B = (101)_2 \& (011)_2 = (001)_2 = 1$

**OR (	):** Bitwise OR is also a binary operator that operates on two equal-length bit patterns, similar to bitwise AND. If both bits in the compared position of the bit patterns are 0, the bit in the resulting bit pattern is 0, otherwise 1.

- $A = 5 = (101)_2$, $B = 3 = (011)_2$ $A | B = (101)_2 | (011)_2 = (111)_2 = 7$

- **XOR (^):** Bitwise XOR also takes two equal-length bit patterns. If both bits in the compared position of the bit patterns are 0 or 1, the bit in the resulting bit pattern is 0, otherwise 1.

  $A = 5 = (101)_2$, $B = 3 = (011)_2$ $A \wedge B = (101)_2 \wedge (011)_2 = (110)_2 = 6$

- **Left Shift («):** Left shift operator is a binary operator which shift the some number of bits, in the given bit pattern, to the left and append 0 at the end. Left shift is equivalent to multiplying the bit pattern with 2k (if we are shifting k bits).

  $1 \ll 1 = 2 = 2^1$ $1 \ll 2 = 4 = 2^2$ $1 \ll 3 = 8 = 2^3$ $1 \ll 4 = 16 = 2^4$ … $1 \ll n = 2^n$

- **Right Shift (»):** Right shift operator is a binary operator which shift the some number of bits, in the given bit pattern, to the right and append 1 at the end. Right shift is equivalent to dividing the bit pattern with 2k (if we are shifting k bits).

  $4 \gg 1 = 2$ $6 \gg 1 = 3$ $5 \gg 1 = 2$ $16 \gg 4 = 1$

X	Y	X&Y	X|Y	X^Y	~(X)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Note:-

Bitwise operators are good for saving space and sometimes to cleverly remove dependencies.

Tricks with Bit

1. **x ^ (x & (x-1)) :** Returns the rightmost 1 in binary representation of x.
   As explained above, (x & (x - 1)) will have all the bits equal to the x except for the rightmost 1 in x.
   So if we do bitwise XOR of x and (x & (x-1)), it will simply return the rightmost 1.
   x = 10 = $(1010)_2$ **x & (x-1)** = $(1010)_2$ & $(1001)_2$ = $(1000)_2$ **x ^ (x & (x-1))** = $(1010)_2$ ^ $(1000)_2$ = $(0010)_2$
2. **x & (-x) :** Returns the rightmost 1 in binary representation of x
   (-x) is the two's complement of x. (-x) will be equal to one's complement of x plus 1. Therefore (-x) will have all the bits flipped that are on the left of the rightmost 1 in x. So x & (-x) will return rightmost 1.
   x = 10 = $(1010)_2$ **(-x)** = -10 = $(0110)_2$ **x & (-x)** = $(1010)_2$ & $(0110)_2$ = $(0010)_2$
3.

**x	(1 « n) :** Returns the number x with the nth bit set.

4. (1 « n) will return a number with only nth bit set. So if we OR it with x it will set the nth bit of x.
5. **x** = 10 = $(1010)_2$ and n = 2 **1 « n** = $(0100)_2$ **x | (1 « n)** = $(1010)_2$ | $(0100)_2$ = $(1110)_2$

Applications of Bit Operations

1. They are widely used in areas of graphics ,specially XOR(Exclusive OR) operations.
2. They are widely used in the embedded systems, in situations, where we need to set/clear/toggle just one single bit of a specific register without modifying the other contents. We can do OR/AND/XOR operations with the appropriate mask for the bit position.
3. Data structure like n-bit map can be used to allocate n-size resource pool to represent the current status.
4. Bits are used in networking, framing the packets of numerous bits which is sent to another system generally through any type of serial interface.

Standard Bit Operations Problems

1. Check for Opposite Sign

Problem:

Given two signed integers, write a function that returns true if the signs of given integers are different, otherwise false.

The function should not use any of the arithmetic operators.

Approach:

Let the given integers be x and y. The sign bit is 1 in negative numbers, and 0 in positive numbers.

The XOR of x and y will have the sign bit as 1 iff they have opposite sign.

In other words, XOR of x and y will be negative number number iff x and y have opposite signs.

Implementation:

```
def check_opposite_sign(x, y):
    return (x^y < 0)


print("Example-1: check_opposite_sign(-2, 7)")
print(check_opposite_sign(-2, 7))

print("Example-2: check_opposite_sign(-2, -7)")
print(check_opposite_sign(-2, -7))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 12:44:59 ~/Personal/Notebooks/Algorithms/8. Bit Algorithms $ python3 1_opposite_sign.py
Example-1: check_opposite_sign(-2, 7)
True
Example-2: check_opposite_sign(-2, -7)
False
```

Complexity:

- **Time: O(1)**
- **Auxilliary Space: O(1)**

2. Divisible by 3

Problem:

Check if a number is divisible by 3.

Naive Approach:

If sum of digits in a number is divisible by 3 then number is divisible by 3 e.g., for 612 sum of digits is 9 so it is divisible by 3.

But this solution is not efficient as we will have to get all decimal digits one by one, add them and then check if sum is divisible by 3.

Efficient Bit Approach:

If difference between count of odd set bits (Bits set at odd positions) and even set bits in binary representation of number is divisible by 3 then number is divisible by 3.

Example:- 23 (00..10111)

Count of all set bits at odd positions = 3

Count of all set bits at even positions = 1

Difference is 2 not divisible by 3 and hence 23 is not divisible by 3.

Algorithm:

- Make n positive if n is negative.
- If number is 0 then return 1
- If number is 1 then return 0
- Initialize: odd_count = 0, even_count = 0
- Loop while n != 0
    - a) If rightmost bit is set then increment odd count.
    - b) Right-shift n by 1 bit
    - c) If rightmost bit is set then increment even count.
    - d) Right-shift n by 1 bit
- return is_divisible_by_3(odd_count - even_count)

Implementation:

```
def is_divisible_by_3(n):
    # Make n positive coz if +n is divisible by 3 then -n is also divisible.
    n = abs(n)

    if(n == 0):
        return True

    if(n == 1):
        return False

    odd_count = 0; even_count = 0
    while(n):
        # If odd bit is set then increment odd counter
```

```
        if(n & 1):
            odd_count += 1
        n = n >> 1

        # If even bit is set then increment even counter
        if(n & 1):
            even_count += 1
        n = n >> 1

    return is_divisible_by_3(abs(odd_count-even_count))



print("Example-1: is_divisible_by_3(23)")
print(is_divisible_by_3(23))

print("Example-2: is_divisible_by_3(111)")
print(is_divisible_by_3(111))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 12:27:11 ~/Personal/Notebooks/Algorithms/8. Bit Algorithms $ python3 2_divisible_by_3.py
Example-1: is_divisible_by_3(23)
False
Example-2: is_divisible_by_3(111)
True
```

Complexity:

- **Time: O(logn)**
- **Auxilliary Space: O(1)**

3. Multiply by 7

Problem:

Efficiently multiply a number by 7.

Approach:

Let the number be n. Then **n*7 = n*(8-1) = n*8 - n**

Algorithm:

(n « 3) - n

Implementation:

```
def multiply_by_7(n):
    print((n<<3) - n)


print("Example-1: multiply_by_7(5)")
multiply_by_7(5)

print("Example-2: multiply_by_7(19)")
multiply_by_7(19)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 12:30:48 ~/Personal/Notebooks/Algorithms/8. Bit Algorithms $ python3 3_multiply_by_7.py
Example-1: multiply_by_7(5)
35
Example-2: multiply_by_7(19)
133
```

Complexity:

- **Time: O(1)**
- **Space: O(1)**

4. Check power of 2***

Problem:

Consider a number N and you need to find if N is a power of 2.

Approach:

Binary representation of (x-1) will have all the bits same as x, except for the rightmost 1 in x and all the bits to the right of the rightmost 1.

$x = 4 = (100)_2$ $x - 1 = 3 = (011)_2$ $x = 6 = (110)_2$ $x - 1 = 5 = (101)_2$

Binary representation of (x-1) can be obtained by simply flipping all the bits to the right of rightmost 1 in x and also including the rightmost 1.

Now: x & (x-1) will have all the bits equal to the x except for the rightmost 1 in x. **x = 4** = $(100)_2$ x - 1 = 3 = $(011)_2$

x & (x-1) = 4 & 3 = $(100)_2$ & $(011)_2$ = **$(000)_2$**

x = 6 = $(110)_2$ x - 1 = 5 = $(101)_2$ x & (x-1) = 6 & 5 = $(110)_2$ & $(101)_2$ = **$(100)_2$**

Properties for numbers which are powers of 2, is that they have one and only one bit set in their binary representation.

If the number is neither zero nor a power of two, it will have 1 in more than one place. So if x is a power of 2 then x & (x-1) will be 0.

Implementation:

```
check_power_of_2(x):
    # x will check if x == 0 and !(x & (x - 1)) will check if x is a power of 2 or not
    return (x && !(x & (x - 1)))
```

5. Count 1's in Binary Representation***

Problem:

Count number of 1's present in binary representation of the given number.

Approach:

In x-1, the rightmost 1 and bits right to it are flipped, then by performing **x&(x-1)**, and storing it in x, will reduce x to a number containing number of ones(in its binary form) less than the previous state of x, thus increasing the value of count in each iteration.

Implementation:

```
count_ones(n):
    while(n):
        n = n&(n-1)
        count += 1

    return count
```

Explanation

Example: n = 23 = {10111}2 .

Initially **n = 23, count = 0**

Now, n will change to n&(n-1)

n-1 = 22 = {10110}2 n&(n-1) = {101112 & {10110}2 = {10110}2 = 22

n = 22, count = 1

n-1 = 21 = {10101}2 n&(n-1) = {10110}2 & {10101}2 = {10100}2 = 20

n = 20, count = 2

n-1 = 19 = {10011}2 n&(n-1) = {10100}2 & {10011}2 = {10000}2 =16

n = 16, count = 3

n-1 = 15 = {01111}2 n&(n-1) = {10000}2 & {01111}2 = {00000}2 = 0

n = 0, count = 4

n = 0, the the loop will terminate and gives the result as 4.

Complexity:

- **Time: O(k)**

6. Check if i^{th} bit is set***

Problem:

Check if i^{th} bit is set or not in a given number.

Approach:

To check whether it's ith bit is set or not, we can **AND** it with the number 2^i .

The binary form of 2^i contains only ith bit as set (or 1), else every bit is 0 there.

When we will AND it with N, and if the ith bit of N is set, then it will return a non zero number (2^i to be specific), else 0 will be returned.

Implementation

```
check_ith_bit_set(N, i):
    if( N & (1 << i) ):
        return True
    else:
        return False;
```

Explanation:

Example: N = 20 = $\{10100\}_2$. Check if it's **2nd bit** is set or not(starting from 0).

For that, we have to AND it with $2^2 = 1«2 = \{100\}_2$ **20 & 4 =** {10100} & {100} = {100} = 22 = 4 (non-zero number)

Which means it's 2nd bit is set.

7. Generate all Possible Subsets of a Set***

Problem:

Given a set generate all its possible subsets.

Approach:

Represent each element in a subset with a bit.

A bit can be either 0 or 1, thus we can use this to denote whether the corresponding element belongs to this given subset or not.

So each bit pattern will represent a subset.

Example:

Consider a set A of 3 elements: **A = {a, b, c}**

Now, we need 3 bits, one bit for each element.

1 represent that the corresponding element is present in the subset, whereas 0 represent the corresponding element is not in the subset.

All the possible combination of these 3 bits.

$0 = (000)_2 = \{\}$ $1 = (001)_2 = \{c\}$ $2 = (010)_2 = \{b\}$ $3 = (011)_2 = \{b, c\}$ $4 = (100)_2 = \{a\}$ $5 = (101)_2 = \{a, c\}$ $6 = (110)_2 = \{a, b\}$ $7 = (111)_2 = \{a, b, c\}$

Implementation:

```
print_possible_subsets(A[]):
    n = len(A)
    N = 1<<n
    for i in range(N):
        for j in range(n):
            if(i & (1 << j)):
                print(A[j], end="")
        print()
```

Mathematical Algorithms

- Algorithms deals with Mathematical calculations and its aspects.

Standard Mathematical Algorithm Problems

1. Lucky Numbers***

What are lucky numbers?

Take the set of integers

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19,……

First, **delete every second number**, we get following reduced set.

1, 3, 5, 7, 9, 11, 13, 15, 17, 19,……

Now, **delete every third number**, we get

1, 3, 7, 9, 13, 15, 19,……..

Continue this process indefinitely……

Any number that does NOT get deleted due to above process is called "lucky".

Therefore, **set of lucky numbers is 1, 3, 7, 13,……**

Problem:

Given a number check if it is lucky or not.

Algorithm:

- Before every iteration, if we calculate position of the given no, then in a given iteration, we can determine if the no will be deleted.
- Suppose calculated position for the given no. is p before some iteration, and each ith number is going to be removed in this iteration,
  - if p < i then input number is lucky,
  - if p is such that p%i == 0 (i is a divisor of p), then input number is not lucky.

Implementation:

```
def check_lucky(n, k):
    # n is current_position of number
    # k represents at this iteration kth numbers will be deleted, at start k = 2

    if(n%k == 0):
        return False

    if(k > n):
        return True

    # Calculate new_position after deleting kth number, increase k
    n = n - int(n/k)
    k += 1

    return check_lucky(n, k)



print("Example-1: check_lucky : 5")
print(check_lucky(5, 2))

print("Example-2: check_lucky : 19")
print(check_lucky(19, 2))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 12:14:37 ~/Personal/Notebooks/Algorithms/9. Mathematical Algorithms $ python3 1_check_lucky_number.py
Example-1: check_lucky : 5
False
Example-2: check_lucky : 19
True
```

Complexity:

- **Time: O(loglogn)**

2. Square Root - Babylonian Method***

Problem:

Find approximate square root of a number.

Algorithm:

1. Start with an arbitrary positive start value x (the closer to the root, the better).
2. Initialize y = 1.
3. Do following until desired approximation is achieved.
    - a) Get the next approximation for root using average of x and y
    - b) Set y = n/x

Implementation:

```
def square_root(n):
    # Here using n itself as initial approximation but this can definitely be improved
    x = n
    y = 1

    # a decides the accuracy level
    a = 0.000001

    while(x - y > a):
        x = (x + y)/2
        y = n / x

    return x


print("Example-1: square_root : 4")
print(round(square_root(4), 5))

print("Example-2: square_root : 50")
print(round(square_root(50), 5))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 12:15:37 ~/Personal/Notebooks/Algorithms/9. Mathematical Algorithms $ python3 2_babylonian_square_root.py
Example-1: square_root : 4
2.0
Example-2: square_root : 50
7.07107
```

Complexity:

- **Time: **

3. Sieve of Eratosthenes

The sieve of Eratosthenes is one of the most efficient ways to find all primes smaller than n when n is smaller than 10 million or so.

Problem:

Given a number n, print all primes smaller than or equal to n. It is also given that n is a small number.

Example:

Input : n = 10

Output : 2 3 5 7

Input : n = 20

Output: 2 3 5 7 11 13 17 19

Algorithm:

1. Create a list of consecutive integers from 2 to n: (2, 3, 4, …, n).
2. Initially, let p=2, the first prime number.
3. Starting from p2, count up in increments of p and mark each of these numbers greater than or equal to p2 itself in the list. These numbers will be p(p+1), p(p+2), p(p+3), etc..
4. Find the first number greater than p in the list that is not marked.
    - If there was no such number, stop.
    - Otherwise, let p now equal this number (which is the next prime), and repeat from step 3.

Implementation:

```
def sieve_of_eratosthenes(n):
    primes = [True]*(n+1)
    p = 2

    # Initialize all to be True, a value in prime[i] will finally be False if i is Not a prime, else True.
    while(p*p <= n):
        # If prime[p] is still True, then it is a prime
        if(primes[p] == True):
            # Update all multiples of p staring from p*p, then p*(p+1), p*(p+2) and so on
            for i in range(p*p, n+1, p):
                primes[i] = False

        p += 1

    # Print all primes
    for i in range(2, n+1):
        if(primes[i]):
            print(i, end=" ")
    print()
```

```
print("Example-1: sieve_of_eratosthenes : 30")
sieve_of_eratosthenes(30)

print("\nExample-2: sieve_of_eratosthenes : 101")
sieve_of_eratosthenes(101)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 12:16:32 ~/Personal/Notebooks/Algorithms/9. Mathematical Algorithms $ python3 3_sieve_of_eratosthenes.py
Example-1: sieve_of_eratosthenes : 30
2 3 5 7 11 13 17 19 23 29

Example-2: sieve_of_eratosthenes : 101
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101
```

Complexity:

- **Time: **

4. Next Smallest Palindrome***

Problem:

Given a number, find the next smallest palindrome larger than this number.

Examples:

Input: 999 Output: 1001

Input: 1234 Output: 1331

Input: 1213 Output: 1221

Input: 1221 Output: 1331

Input: 23921 Output: 23932

Input: 23941 Output: 24042

Implementation:

```
import math
```

```
def reverse(string):
    return string[::-1]


def all_9s(n):
    for digit in n:
        if(digit != "9"):
            return False
    return True


def next_smallest_palindrome(n):
    n = str(n)
    k = len(n)
    # If all digits are 9s then put 1(k-1 zeroes)1
    if(all_9s(n)):
        return "1"+str("0"*(k-1))+"1"

    # Get the left half
    left_half = n[:(int(math.ceil(k/2)))]
    # If the number of digits are even
    if(k%2==0):
        # Check if created palindrome i.e. pal = left_half + reverse(left_half)
        # is lesser or equal to given number
        # If it is lesser or equal increment left_half by 1.
        if(left_half + reverse(left_half) <= n):
            left_half = str(int(left_half)+1)

        pal = left_half + reverse(left_half)
    # If number of digits are odd
    else:
        if(left_half + reverse(left_half[:-1]) <= n):
            left_half = str(int(left_half)+1)

        pal = left_half + reverse(left_half[:-1])

    return pal


print("Example-1: next_smallest_palindrome(999)")
print(next_smallest_palindrome(999))

print("Example-2: next_smallest_palindrome(1234)")
print(next_smallest_palindrome(1234))

print("Example-3: next_smallest_palindrome(1213)")
print(next_smallest_palindrome(1213))

print("Example-4: next_smallest_palindrome(1221)")
print(next_smallest_palindrome(1221))

print("Example-5: next_smallest_palindrome(23921)")
print(next_smallest_palindrome(23921))

print("Example-6: next_smallest_palindrome(23941)")
```

```
print(next_smallest_palindrome(23941))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 16:17:08 ~/Personal/Notebooks/Algorithms/9. Mathematical Algorithms $ python3 4_next_smallest_palindrome.py
Example-1: next_smallest_palindrome(999)
1001
Example-2: next_smallest_palindrome(1234)
1331
Example-3: next_smallest_palindrome(1213)
1221
Example-4: next_smallest_palindrome(1221)
1331
Example-5: next_smallest_palindrome(23921)
23932
Example-6: next_smallest_palindrome(23941)
24042
```

Complexity:

- **Time: O(n) **

5. Prime Factors

Problem:

Given a number n, write an efficient function to print all prime factors of n.

Examples:

Input: 12 Output: 2, 2, 3

Input: 315 Output: 3, 3, 5, 7

Algorithm:

1. While n is divisible by 2, print 2 and divide n by 2.
2. After step 1, n must be odd. Now start a loop from i = 3 to square root of n. While i divides n, print i and divide n by i, increment i by 2 and continue.
3. If n is a prime number and is greater than 2, then n will not become 1 by above two steps. So print n if it is greater than 2.

Implementation

```
import math

def prime_factors(n):
    # Check 2 as prime factors
    while(n%2 == 0):
        print(2, end=" ")
        n = int(n/2)

    # Check 3, 5, 7, 11, ... and so on as prime factors
    for i in range(3, int(math.sqrt(n))+1, 2):
        while(n%i==0):
            print(i, end=" ")
            n = int(n/i)


    # If n is a prime greater than 2
    if(n > 2):
        print(n)

    print()


print("Example-1: prime_factors(12)")
prime_factors(12)

print("Example-2: prime_factors(315)")
prime_factors(315)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 20:15:48 ~/Personal/Notebooks/Algorithms/9. Mathematical Algorithms $ python3 5_prime_factors.py
Example-1: prime_factors(12)
2 2 3

Example-2: prime_factors(315)
3 3 5 7
```

Complexity:

- **Time: O(n) **

6. Trailing Zeroes in Factorial

Problem:

Given an integer n, write a function that returns count of trailing zeroes in n!.

Examples:

Input: 5 Output: 1 coz 5! = 120 1 trailing zero

Input: 20 Output: 4 coz 20! = 2432902008176640000 4 trailing zeroes

Input: 100 Output: 24

Approach:

Trailing 0s in n! = Count of 5s in prime factors of n! = floor(n/5) + floor(n/25) + floor(n/125) + ….

Implementation:

```
def factorial_trailing_zeroes(n):
    count = 0
    k = 5
    while(n > 0):
        n = int(n/k)
        count += n
        k = k*k

    print(count)



print("Example-1: factorial_trailing_zeroes(5)")
factorial_trailing_zeroes(5)

print("Example-2: factorial_trailing_zeroes(20)")
factorial_trailing_zeroes(20)

print("Example-3: factorial_trailing_zeroes(100)")
factorial_trailing_zeroes(100)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 22:42:30 ~/Personal/Notebooks/Algorithms/9. Mathematical Algorithms $ python3 6_factorial_trailing_zeroes.py
Example-1: factorial_trailing_zeroes(5)
1
Example-2: factorial_trailing_zeroes(20)
4
Example-3: factorial_trailing_zeroes(100)
20
```

Complexity:

- **Time: O(loglogn) **

7. Next Greater with Same Digits***

Problem:

Given a number n, find the smallest number that has same set of digits as n and is greater than n.

If n is the greatest possible number with its set of digits, then print “not possible”.

Examples:

Input: 218765 Output: 251678

Input: 1234 Output: 1243

Input: 4321 Output: “Not Possible”

Input: 534976 Output: 536479

Algorithm:

- Traverse the given number from rightmost digit, keep traversing till you find a digit which is smaller than the previously traversed digit.
  - **Example:** if the input number is “534976”, we stop at **4** because 4 is smaller than next digit 9.
  - If we do not find such a digit, then output is “Not Possible”.
- Now search the right side of above found digit ‘d’ for the smallest digit greater than ‘d’.
  - **Example:** In “53**4**976″, the right side of 4 contains “976”. The smallest digit greater than 4 is **6**.
- Swap the above found two digits, we get 53**6**97**4** in above example.
- Now sort all digits from position next to ‘d’ to the end of number. The number that we get after sorting is the output.
  - **Example:** In above, we sort digits in bold 536**974**. We get “536**479**” which is the next greater number for input 534976.

Implementation:

```
def next_greater_same_digits(n):
    n = list(str(n))
    k = len(n)

    # Finding a  digit which is smaller than the previously traversed digit
    for i in range(k-1, -1, -1):
        if(n[i-1] < n[i]):
            break

    if(i==0):
        return "Not possible"
```

```
    # Find the smallest digit on the right side of (i-1)'th digit that is greater than x
    x = n[i-1]
    smallest = i
    for j in range(i+1, k):
        if(n[j] < n[smallest] and  n[j] > x):
            smallest = j

    # Swapping the above found smallest digit with (i-1)'th
    n[i-1], n[smallest] = n[smallest], n[i-1]

    # Sort all the digits from ith digit and concatenate it with left_part
    result_num = n[:i] + sorted(n[i:])

    # Join all the array element to create a number
    result_num = "".join(result_num)

    return result_num



print("Example-1: next_greater_same_digits(218765)")
print(next_greater_same_digits(218765))

print("\nExample-2: next_greater_same_digits(1234)")
print(next_greater_same_digits(1234))

print("\nExample-3: next_greater_same_digits(4321)")
print(next_greater_same_digits(4321))

print("\nExample-4: next_greater_same_digits(534976)")
print(next_greater_same_digits(534976))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 22:36:46 ~/Personal/Notebooks/Algorithms/9. Mathematical Algorithms $ python3 7_next_greater_same_digits.py
Example-1: next_greater_same_digits(218765)
251678

Example-2: next_greater_same_digits(1234)
1243

Example-3: next_greater_same_digits(4321)
Not possible

Example-4: next_greater_same_digits(534976)
536479
```

Complexity:

- **Time: O(size)**

8. Clock Angle Problem***

Problem:

Find angle between hands of an analog clock at a given time.

Examples:

Input: h = 9:00, m = 60.00 Output: 90 degree

Input: h = 12:00, m = 30.00 Output: 165 degree

Input: h = 3.00, m = 30.00 Output: 75 degree

Approach:

The idea is to take 12:00 (h = 12, m = 0) as a reference. Then

1. Calculate the angle made by hour hand with respect to 12:00 in h hours and m minutes.
2. Calculate the angle made by minute hand with respect to 12:00 in h hours and m minutes.
3. The difference between two angles is the angle between two hands.

Calculation:

- **Hour Hand:**
  - In 12 hours 360º, then 1hour = 30º
  - In 720 minutes 360º, then 1min = 0.5º
  - **Angle travelled by Hour Hand = (h*30) + (m*0.5)**
- **Minute Hand:**
  - In 60 min 360º, then 1min = 6º
  - **Angle travelled by Minute Hand = (m*6)**

Implementation:

```
def clock_angle(hours, minutes):
    h = hours%12
    m = minutes%60

    hour_hand_angle = h*30 + m*0.5
```

```
    minute_hand_angle = m*6

    angle = abs(hour_hand_angle-minute_hand_angle)

    return min(angle, 360-angle)



print("Example-1: clock_angle(9, 60)")
print(clock_angle(9, 60))

print("Example-2: clock_angle(12, 30)")
print(clock_angle(12, 30))

print("Example-3: clock_angle(3, 30)")
print(clock_angle(3, 30))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 00:40:06 ~/Personal/Notebooks/Algorithms/9. Mathematical Algorithms $ python3 9_clock_angle.py
Example-1: clock_angle(9, 60)
90.0
Example-2: clock_angle(12, 30)
165.0
Example-3: clock_angle(3, 30)
75.0
```

Complexity:

- **Time: O(1)**

9. Smallest number whose digit multiply to give number N

Problem:

Given a number ‘n’, find the smallest number ‘p’ such that if we multiply all digits of ‘p’, we get ‘n’.

The result ‘p’ should have minimum two digits.

Examples:

Input: n = 36 Output: p = 49 // Note that 4*9 = 36 and 49 is the smallest such number

Input: n = 100 Output: p = 455 // Note that 455 = 100 and 455 is the smallest such number

Input: n = 7 Output: p = 17 // Note that 1*7 = 7

Input: n = 13 Output: Not Possible

Approach:

- **Case 1: n < 10**
  - When n is smaller than n, the output is always n+10.
  - Example: n = 7, output is 17. For n = 9, output is 19.
- **Case 2: n >= 10**
  - Find all factors of n which are between 2 and 9 (both inclusive).
  - The idea is to start searching from 9 so that the number of digits in result are minimized.
  - Example: 9 is preferred over 33 and 8 is preferred over 24.
  - Store all found factors in an array, the array would contain digits in non-increasing order, so finally print the array in reverse order.

Implementation:

```
def smallest_num_digit_multiply_to_n(n):
    if(n<10):
        return 10+n

    # p is to store the digits
    p = ""
    for i in range(9, 1, -1):
        while(n%i==0):
            p += str(i)
            n = int(n/i)

    # If n is still grteater than 10
    if(n>10):
        return "Not Possible"

    # Reverse the p now.
    p = p[::-1]

    return p


print("Example-1: smallest_num_digit_multiply_to_n(36)")
print(smallest_num_digit_multiply_to_n(36))

print("Example-2: smallest_num_digit_multiply_to_n(100)")
print(smallest_num_digit_multiply_to_n(100))

print("Example-3: smallest_num_digit_multiply_to_n(7)")
print(smallest_num_digit_multiply_to_n(7))

print("Example-3: smallest_num_digit_multiply_to_n(13)")
print(smallest_num_digit_multiply_to_n(13))
```

Output:

```
astikanand@Developer-MAC 03:21:48 ~/Interview-Preparation/Algorithms/9. Mathematical Algorithms $ python3 10_smallest_num_digit_multiply_to_n.py
Example-1: smallest_num_digit_multiply_to_n(36)
49
Example-2: smallest_num_digit_multiply_to_n(100)
455
Example-3: smallest_num_digit_multiply_to_n(7)
17
Example-3: smallest_num_digit_multiply_to_n(13)
Not Possible
```

Complexity:

- **Time: O(1)**

10. Birthday Paradox***

Q1. How many people must be there in a room to make the probability 100% that at-least two people in the room have same birthday?

Answer: 367 (since there are 366 possible birthdays, including February 29).

The above question was simple, let us see the below question.

Q2. How many people must be there in a room to make the probability 50% that at-least two people in the room have same birthday?

Answer: 23

The number is surprisingly very low. In fact, we need only 70 people to make the probability 99.9 %.

Generalized Formula

What is the probability that two persons among n have same birthday?

P(same) : The probability that 2 people among n have same birthday.

P(same) = 1 - P(different)

→ Probability that all of them have different birthday.

$$P(\text{different}) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots\cdots\cdots \times \frac{(n-(n-1))}{365}$$

- $\frac{365}{365}$: 1st person can have any birthday among 365 days
- $\frac{364}{365}$: 2nd person should have a birthday which is not same as 1st person
- $\frac{363}{365}$: 3rd person should have a birthday which is not same as first 2 persons.
- $\frac{(n-(n-1))}{365}$: nth person should have a birthday which is not same as any of previous n-1 persons.

We know,

$$e^x = 1 + x + \frac{x^2}{2!} + \cdots\cdots$$

$$e^x \approx 1 + x \qquad \text{(first order approximation when } x \ll 1\text{)}$$

↓ We can write,

$$e^{\frac{-a}{365}} \approx 1 - \frac{a}{365}$$

Now,

$$P(\text{different}) = 1 \times \left(1 - \frac{1}{365}\right) \times \left(1 - \frac{2}{365}\right) \cdots\cdots\cdots \times \left(1 - \frac{n-1}{365}\right)$$

$$\approx 1 \times e^{\frac{-1}{365}} \times e^{\frac{-2}{365}} \times \cdots\cdots\cdots \times e^{\frac{-(n-1)}{365}}$$

$$\approx e^{\frac{-(1+2+\cdots\cdots+(n-1))}{365}}$$

$$\approx e^{\frac{-(n(n-1))}{2\times365}} \approx e^{\frac{-n^2}{2\times365}}$$

P(same) = 1 - P(different)

$$P(\text{same}) = 1 - e^{\frac{-n^2}{2\times365}}$$

Taking log on both sides we get,

$$n \approx \sqrt{2 \times 365 \ln\left(\frac{1}{1 - P(\text{same})}\right)}$$

→ Can find how much n is required for filling a particular probability value.

Examples:- p = 50% n = 23

p = 70% n = 30

Implementation:

```
import math

def birthday_paradox_find_n(probability):
    print (math.ceil(math.sqrt(2*365*(math.log(1/(1-probability))))))


print ("Example-1: birthday_paradox_find_n(0.50)")
birthday_paradox_find_n(0.50)

print ("Example-2: birthday_paradox_find_n(0.70)")
birthday_paradox_find_n(0.70)

print ("Example-1: birthday_paradox_find_n(0.99)")
birthday_paradox_find_n(0.99)

print ("Example-1: birthday_paradox_find_n(0.999)")
birthday_paradox_find_n(0.999)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 03:25:41 ~/Personal/Notebooks/Algorithms/9. Mathematical Algorithms $ python3 11_birthday_paradox.py
Example-1: birthday_paradox_find_n(0.50)
23
Example-2: birthday_paradox_find_n(0.70)
30
Example-1: birthday_paradox_find_n(0.99)
58
Example-1: birthday_paradox_find_n(0.999)
72
```

Complexity:

- **Time: O(1)**

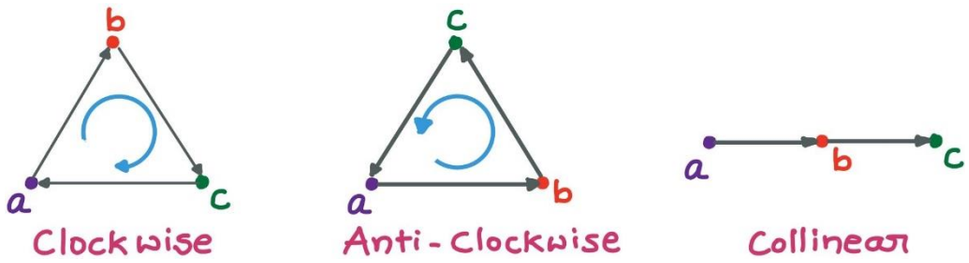
Geometric Algorithms

These algorithms are designed to solve Geometric Problems. They requires in-depth knowledge of different mathematical subjects like combinatorics, topology, algebra, differential geometry etc.

Concept of Orientation

Orientation of an ordered triplet of points in the plane can be:

- Clockwise
- Anti-clockwise
- Collinear



Orientation Formula

$$(b_y - a_y)\,(c_x - b_x) - (b_x - a_x)\,(c_y - b_y)$$

> 0 : Clockwise **= 0 :** Collinear **< 0 :** Anti-Clockwise

Standard Geometric Algorithm Problems

1. Check if 2 line segements intersect***

Problem:

Given two line segments (p1, q1) and (p2, q2), find if the given line segments intersect with each other.

Examples:

Input: **Line-1:** p1 = {1, 1}, q1 = {10, 1} **Line-2:** p1 = {1, 2}, q1 = {10, 2}

Output: No

Input: **Line-1:** p1 = {10, 0}, q1 = {0, 10} **Line-2:** p1 = {0, 0}, q1 = {10, 10}

Output: Yes

Input: **Line-1:** p1 = {-5, -5}, q1 = {0, 0} **Line-2:** p1 = {1, 1}, q1 = {10, 10}

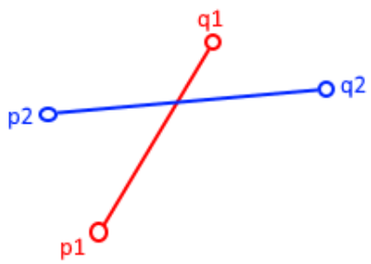
Output: No

Intersection of 2 line segments

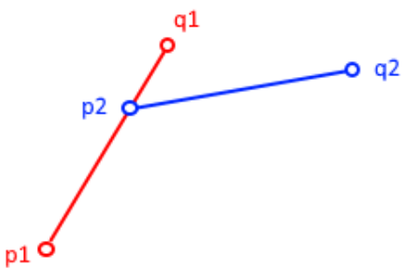
Two segments (p1,q1) and (p2,q2) intersect if and only if one of the following two conditions is verified:

Case-1: General Case

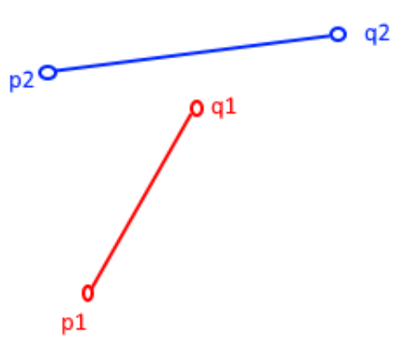
- (p1, q1, p2) and (p1, q1, q2) have different orientations and
- (p2, q2, p1) and (p2, q2, q1) have different orientations.



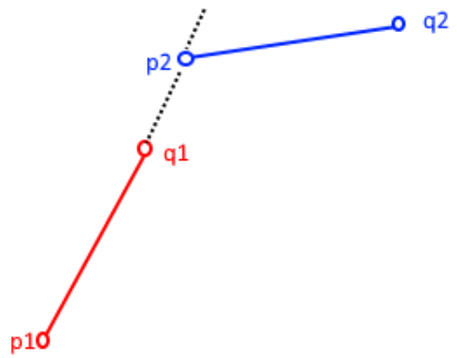
Example : Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) also differnet.



Example: Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) also different



Example: Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) are same

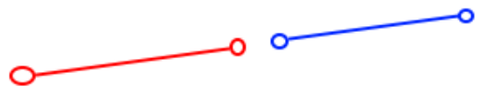


Example: Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) are same.

Case-2: Special Case

- (p1, q1, p2), (p1, q1, q2), (p2, q2, p1), and (p2, q2, q1) are all collinear and
- the x-projections of (p1, q1) and (p2, q2) intersect
- the y-projections of (p1, q1) and (p2, q2) intersect

Example: All points are collinear. The x-projections pf (p1,q1) and (p2,q2) intersect. The y-projection of (p1,q1) and(p2,q2) also intersect



Example: All points are collinear .The x-projections of (p1,q1) and (p2,q2) not intersect. The y-projection of (p1,q1) and (p2,q2) do not intersect

Implementation:

```
def orientation(a, b, c):
    val = (b[1]-a[1])*(c[0]-b[0]) - (b[0]-a[0])*(c[1]-b[1])

    if val == 0:
        return 0
    elif val > 0:
        return 1
    else:
        return -1


def on_segment(a, b, c):
    if (b[0] <= max(a[0], c[0]) and b[0] >= min(a[0], c[0]) and
        b[1] <= max(a[1], c[1]) and b[1] >= min(a[1], c[1])):
        return True

    return False


def check_2_line_segments_intersection(line1, line2):
    p1 = line1[0]; q1 = line1[1]
    p2 = line2[0]; q2 = line2[1]

    # All 4 different orientation
    #   • p1, q1, p2 = o1 and p1, q1, q2 = o2 --> Should be different
    #   • p2, q2, p1 = o3 and p2, q2, q1 = o4 --> Should be different
    o1 = orientation(p1, q1, p2)
    o2 = orientation(p1, q1, q2)
    o3 = orientation(p2, q2, p1)
    o4 = orientation(p2, q2, q1)
```

```
    result = False

    ### General Case
    if o1 != o2 and o3 != o4:
        result = True

    ### Case-2: Special Cases
    # p1, q1 and p2 are collinear and p2 lies on segment p1q1
    if o1 == 0 and on_segment(p1, p2, q1):
        result = True

    # p1, q1 and q2 are collinear and q2 lies on segment p1q1
    if o2 == 0 and on_segment(p1, q2, q1):
        result = True

    # p2, q2 and p1 are collinear and p1 lies on segment p2q2
    if o3 == 0 and on_segment(p2, p1, q2):
        result = True

    # p2, q2 and q1 are collinear and q1 lies on segment p2q2
    if o4 == 0 and on_segment(p2, q1, q2):
        result = True

    if(result):
        print("Yes")
    else:
        print("No")


print("Example-1: check_2_line_segments_intersection([(1,1), (10,1)], [(1,2), (10,2)])")
check_2_line_segments_intersection([(1,1), (10,1)], [(1,2), (10,2)])

print("\nExample-2: check_2_line_segments_intersection([(10,0), (0,10)], [(0,0), (10,10)])")
check_2_line_segments_intersection([(10,0), (0,10)], [(0,0), (10,10)])

print("\nExample-3: check_2_line_segments_intersection([(-5,-5), (0,0)], [(1,1), (10,10)])")
check_2_line_segments_intersection([(-5,-5), (0,0)], [(1,1), (10,10)])
```

Output:

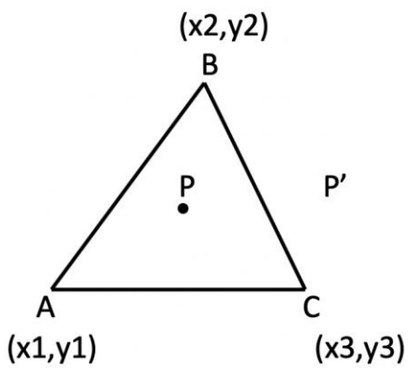
```
astik.anand@mac-C02XD95ZJG5H 21:52:20 ~/Personal/Notebooks/Algorithms/10. Geometric Algorithms $ python3 1_check_2_line_segments_intersect.py
Example-1: check_2_line_segments_intersection([(1,1), (10,1)], [(1,2), (10,2)])
No

Example-2: check_2_line_segments_intersection([(10,0), (0,10)], [(0,0), (10,10)])
Yes

Example-3: check_2_line_segments_intersection([(-5,-5), (0,0)], [(1,1), (10,10)])
No
```

Complexity:

- **Time: O(1)**
- **Auxilliary Space: O(1)**

2. Check if Point lies inside triangle

Problem:
Given three corner points of a triangle, and one more point P. Check if the p lies inside the triangle.



Area of Triangle = $\frac{1}{2}(x1(y2-y3)+x2(y3-y1)+x3(y1-y2))$

Approach: Simple

- Get the area of ΔABC and check if it equals ΔPAB + ΔPAC + ΔPBC.
- If they equal then point P lies inside the triangle.

Implementation:

```
def triangle_area(x1, y1, x2, y2, x3, y3):
   return abs(0.5*(x1*(y2-y3) + x2*(y3-y1)+ x3*(y1-y2)))


def point_lies_inside_triangle(x1, y1, x2, y2, x3, y3, x, y):
    # Calculate Areas
    area_ABC = triangle_area(x1, y1, x2, y2, x3, y3)
    area_PAB = triangle_area(x, y, x1, y1, x2, y2)
```

```
    area_PAC = triangle_area(x, y, x1, y1, x3, y3)
    area_PBC = triangle_area(x, y, x2, y2, x3, y3)

    if(area_ABC == area_PAB + area_PAC + area_PBC):
        print("Inside")
    else:
        print("Outside")



print("Example-1: point_lies_inside_triangle(0, 0, 10, 30, 20, 0, 10, 15)")
point_lies_inside_triangle(0, 0, 10, 30, 20, 0, 10, 15)

print("\nExample-2: point_lies_inside_triangle(0, 0, 10, 30, 20, 0, 18, 25)")
point_lies_inside_triangle(0, 0, 10, 30, 20, 0, 15, 25)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 22:49:20 ~/Personal/Notebooks/Algorithms/10. Geometric Algorithms $ python3 2_point_lies_inside_triangle.py
Example-1: point_lies_inside_triangle(0, 0, 10, 30, 20, 0, 10, 15)
Inside

Example-2: point_lies_inside_triangle(0, 0, 10, 30, 20, 0, 18, 25)
Outside
```

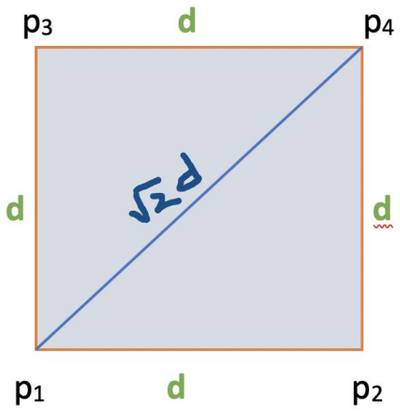
Complexity:

- **Time: O(1)**
- **Auxilliary Space: O(1)**

3. Check if 4 Points form Square***

Problem:

Given coordinates of four points in a plane, find if the four points form a square or not.



Distance b/w 2 points:

$$\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

Approach:

- Pick any point and calculate its distance from the rest of the points. Let the picked point be ‘p’.
- To form a square, the distance of two points must be the same from ‘p’, let this distance be d.
- The distance from third point must be equal to **√2d**. Let this point with different distance be ‘q’.
- The above condition is not good enough as the point with different distance can be on the other side.
- Check that q is at the same distance from 2 other points and this distance is the same as d.

Implementation:

```
def square_distance(a, b):
    return (b[0]-a[0])*(b[0]-a[0]) + (b[1]-a[1])*(b[1]-a[1])


def check_square(p1, p2, p3, p4):
    is_square = False

    ## Calculate distances from p1
    d2 = square_distance(p1, p2) # from p1 to p2
    d3 = square_distance(p1, p3) # from p1 to p3
    d4 = square_distance(p1, p4) # from p1 to p4

    # If lengths of (p1, p2) and (p1, p3) are same, then following conditions must meet to form a square:
    #   • 1) Square of length of (p1, p4) is same as twice the square of (p1, p2)
    #   • 2) Square of length of (p2, p3) is same as twice the square of (p2, p4)

    if (d2 == d3 and 2*d2 == d4 and 2*square_distance(p2, p4) == square_distance(p2, p3)):
        is_square = True

    # Cases similar to above case
    if (d3 == d4 and 2*d3 == d2 and 2*square_distance(p3, p2) == square_distance(p3, p4)):
        is_square = True
```

```
    if (d2 == d4 and 2*d2 == d3 and 2*square_distance(p2, p3) == square_distance(p2, p4)):
        is_square = True

    if(is_square):
        print("Square")
    else:
        print("Not Square")


print("Example-1: check_square((20, 10), (10, 20), (20, 20), (10, 10))")
check_square((20, 10), (10, 20), (20, 20), (10, 10))

print("\nExample-2: check_square((20, 10), (10, 20), (20, 20), (10, 20))")
check_square((20, 10), (10, 20), (20, 20), (10, 20))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 00:19:37 ~/Personal/Notebooks/Algorithms/10. Geometric Algorithms $ python3 3_check_square.py
Example-1: check_square((20, 10), (10, 20), (20, 20), (10, 10))
Square

Example-2: check_square((20, 10), (10, 20), (20, 20), (10, 20))
Not Square
```

Complexity:

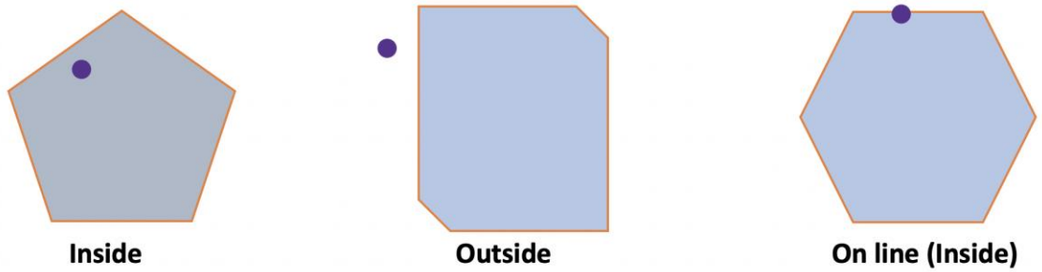
- **Time: O(1)**
- **Auxilliary Space: O(1)**

4. Check if Point lies inside Polygon***

Problem:

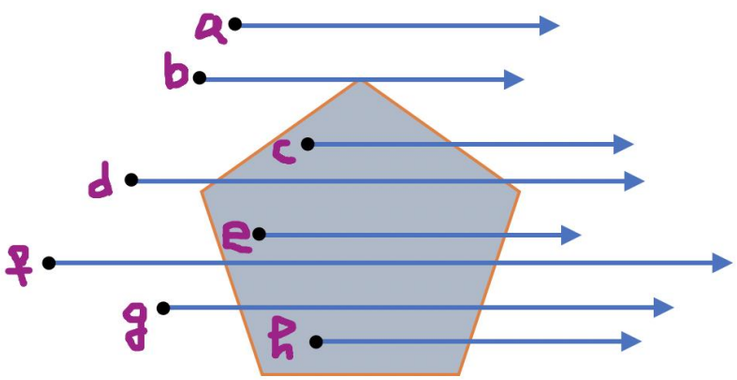
Given a polygon and a point p, check if p lies inside the polygon or not.

The points lying on the border are considered inside.



Approach:

- Draw a horizontal line to the right of each point and extend it to infinity.
- Count the number of times the line intersects with polygon edges.
- A point is inside the polygon if either count of intersections is odd or point lies on an edge of polygon.
- If none of the conditions is true, then point lies outside.



Handling cases such as of point b

- **Note:** We need to return true if the point lies on the line or same as one of the vertices of the given polygon.
- To handle this, after checking if the line from **'p' to extreme** intersects:
  - Check whether **'p' is collinear** with vertices of current line of polygon.
  - If it is collinear, then we check if the point 'p' lies on current side of polygon, if it lies, we return true, else false.

Implementation:

```
import sys
```

```python
def orientation(a, b, c):
    val = (b[1]-a[1])*(c[0]-b[0]) - (b[0]-a[0])*(c[1]-b[1])

    if val == 0:
        return 0
    elif val > 0:
        return 1
    else:
        return -1


def on_segment(a, b, c):
    if (b[0] <= max(a[0], c[0]) and b[0] >= min(a[0], c[0]) and
        b[1] <= max(a[1], c[1]) and b[1] >= min(a[1], c[1])):
        return True

    return False


def check_2_line_segments_intersection(line1, line2):
    p1 = line1[0]; q1 = line1[1]
    p2 = line2[0]; q2 = line2[1]

    # All 4 different orientation
    #   • p1, q1, p2 = o1 and p1, q1, q2 = o2 --> Should be different
    #   • p2, q2, p1 = o3 and p2, q2, q1 = o4 --> Should be different
    o1 = orientation(p1, q1, p2)
    o2 = orientation(p1, q1, q2)
    o3 = orientation(p2, q2, p1)
    o4 = orientation(p2, q2, q1)

    result = False

    ### General Case
    if o1 != o2 and o3 != o4:
        result = True

    ### Case-2: Special Cases
    # p1, q1 and p2 are collinear and p2 lies on segment p1q1
    if o1 == 0 and on_segment(p1, p2, q1):
        result = True

    # p1, q1 and q2 are collinear and q2 lies on segment p1q1
    if o2 == 0 and on_segment(p1, q2, q1):
        result = True

    # p2, q2 and p1 are collinear and p1 lies on segment p2q2
    if o3 == 0 and on_segment(p2, p1, q2):
        result = True

    # p2, q2 and q1 are collinear and q1 lies on segment p2q2
    if o4 == 0 and on_segment(p2, q1, q2):
        result = True

    return result
```

```
def check_point_inside_polygon(polygon, p):
    n = len(polygon)
    # Not a polynomial
    if n < 3:
        print("Outside")
        return

    # Extreme point will have x as "infinity" and y same as point p
    extreme = (sys.maxsize, p[1])

    # Count for counting intersections
    count = 0

    result = False
    for i in range(n):
        current_vertex = polygon[i]
        next_vertex = polygon[(i+1)%n]

        # Check if the line segment from 'p' to 'extreme' intersects
        # with the line segment polygon's current_vertex to next_vertex
        if check_2_line_segments_intersection((current_vertex,next_vertex), (p,extreme)):
            # If the point 'p' is collinear with line segment current_vertex---next_vertex
            # Check if it lies on segment return true if it lies, otherwise false
            if (orientation(current_vertex, p, next_vertex) == 0) :
                result = on_segment(current_vertex, p, next_vertex)
                count = 0
                break
            count += 1

    if(result or count%2 == 1):
        print("Inside")
    else:
        print("Outside")



polygon1 = [(0, 0), (10, 0), (10, 10), (0, 10)]
polygon2 = [(0, 0), (5, 5), (5, 0)]
print("Example-1: check_point_inside_polygon(polygon1, (20, 20))")
check_point_inside_polygon(polygon1, (20, 20))

print("\nExample-2: check_point_inside_polygon(polygon1, (5, 5))")
check_point_inside_polygon(polygon1, (5, 5))

print("\nExample-3: check_point_inside_polygon(polygon1, (-1, 10))")
check_point_inside_polygon(polygon1, (-1, 10))

print("\nExample-4: check_point_inside_polygon(polygon2, (3, 3))")
check_point_inside_polygon(polygon2, (3, 3))

print("\nExample-5: check_point_inside_polygon(polygon2, (5, 1))")
check_point_inside_polygon(polygon2, (5, 1))

print("\nExample-6: check_point_inside_polygon(polygon2, (8, 1))")
```

```
check_point_inside_polygon(polygon2, (8, 1))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 01:52:11 ~/Personal/Notebooks/Algorithms/10. Geometric Algorithms $ python3 4_check_point_inside_polygon.py
Example-1: check_point_inside_polygon(polygon1, (20, 20))
Outside

Example-2: check_point_inside_polygon(polygon1, (5, 5))
Inside

Example-3: check_point_inside_polygon(polygon1, (-1, 10))
Outside

Example-4: check_point_inside_polygon(polygon2, (3, 3))
Inside

Example-5: check_point_inside_polygon(polygon2, (5, 1))
Inside

Example-6: check_point_inside_polygon(polygon2, (8, 1))
Outside
```

Complexity:

- **Time: O(n)** : Size of polygon is n.
- **Auxilliary Space: O(1)**

Randomized Algorithms

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.

Examples:

- ***Randomized Quick Sort:*** Uses random number to pick the next pivot (or randomly shuffles the array).
- ***Karger's algorithm:*** Randomly picks an edge.

Concept of Expectation

A random variable can take any possible value in its range, so it is important to define the **expected value** of any random variable.

Expected Value

Expected value of a discrete random variable is R defined as following:

Suppose R can take value r_1 with probability p_1, value r_2 with probability p_2, and so on, up to value r_k with probability p_k.

Then the expectation of this random variable R is defined as:

$$E[R] = r_1*p_1 + r_2*p_2 + \ldots r_k*p_k$$

Example:

Given a fair dice with 6 faces, the dice is thrown n times, find expected value of sum of all results.

Let's take n = 2, then there are total 36 possible outcomes.

(1, 1), (1, 2), ... (1, 6)
(2, 1), (2, 2), ... (2, 6)
.....
.....
(6, 1), (6, 2), ... (6, 6)

Expected Value of sum of two dice throws

2*(1/36) + 3*(1/36) + ... + 7*(1/36) +
3*(1/36) + 4*(1/36) + ... + 8*(1/36) +
.....
.....
7*(1/36) + 8*(1/36) + ... + 12*(1/36)
= 7

Notes:

- The above way to solve the problem becomes difficult when there are more dice throws.
- If we know about **linearity of expectation**, then we can quickly solve the above problem for any number of throws.

Linearity of Expectation

Let R_1 and R_2 be two discrete random variables on some probability space, then expected value of $R_1 + R_2$ is:

$$E[R_1 + R_2] = E[R_1] + E[R_2]$$

Using Linearity of Expectation

Expected Value of sum of **2 dice throws** = 2*(Expected value of one dice throw)
= 2*(1/6 + 2/6 + ... 6/6)
= 2*7/2
= **7**

Expected value of sum for **n dice throws** = n* 7/2 = **3.5 * n**

Properties of Linearity of Expectation:

1. Linearity of expectation holds for both dependent and independent events.
   But the rule $\mathbf{E[R_1R_2] = E[R_1] * E[R_2]}$ is true only for independent events.

2. Linearity of expectation holds for any number of random variables on some probability space.
   Let R_1, R_2, R_3, ... R_k be k random variables, then
   $\mathbf{E[R_1 + R_2 + R_3 + ... + R_k] = E[R_1] + E[R_2] + E[R_3] + ... + E[R_k]}$

Example:- Hat-Check Problem

Let there be group of n men where every man has one hat. The hats are redistributed and every man gets a random hat back.
What is the expected number of men that get their original hat back.

Solution:
Let R_i be a random variable, the value of random variable is 1 if i'th man gets the same hat back, otherwise 0.
So the expected number of men to get the right hat back is
= $E[R_1] + E[R_2] + .. + E[R_n]$
= $P(R_1 = 1) + P(R_2 = 1) + ... + P(R_n = 1)$ # [Here $P(R_i = 1)$ indicates probability that R_i is 1]
= 1/n + 1/n + ... + 1/n
= 1
So on average 1 person gets the right hat back.

Note: Linearity of expectation is useful for evaluating expected time complexity of randomized algorithms like randomized quick sort.

Probability and Expectation

Number of Trials until Success

If probability of success is p in every trial, then expected number of trials until success is 1/p.

Proof: Let R be a random variable that indicates number of trials until success.

The expected value of R is sum of following infinite series

$E[R] = 1*p + 2*(1-p)*p + 3*(1-p)^2*p + 4*(1-p)^3*p + ...$

Taking **'p'** out

$E[R] = p[1 + 2*(1-p) + 3*(1-p)^2 + 4*(1-p)^3 + ...] ---->(1)$

Multiplying both sides with **'(1-p)'** and subtracting

$(1-p)*E[R] = p[1*(1-p) + 2*(1-p)^2 + 3*(1-p)^3 + ...] --->(2)$

Subtracting (2) from (1), we get

$p*E[R] = p[1 + (1-p) + (1-p)^2 + (1-p)^3 + ...]$

Cancelling p from both sides

$E[R] = [1 + (1-p) + (1-p)^2 + (1-p)^3 + ...]$

Above is an infinite geometric progression with ratio (1-p).
Since (1-p) is less than 1, we can apply sum formula.

$E[R] = 1/[1 - (1-p)] = 1/p$

$E[R] = 1/p$

Example:- Boys/Girls ratio Puzzle

In a country, all families want a boy. They keep having babies till a boy is born.
What is the expected ratio of boys and girls in the country?

Solution:

We can use above result to solve the puzzle.
Probability of success in every trial is 1/2 (assuming girls and boys are equally likely).

Let p be probability of having a baby boy.
Number of kids until a baby boy is born = 1/p = 1/(1/2) = 2
Since expected number of kids in a family is 2, ratio of boys and girls is **50:50**.

Example:- Coupon Collector Puzzle

Suppose there are n types of coupons in a lottery and each lot contains a coupon (with probability 1/n each).
How many lots have to be bought (in expectation) until we have at least one coupon of each type.

Solution:

Let X_i be the number of lots bought before **i'th** new coupon is collected.
Note that X_1 is **1** as the first coupon is always a new coupon (not collected before).

Let 'p' be probability that 2nd coupon is collected in next buy, then value of p is (n-1)/n.
So the number of trials needed before 2nd new coupon is picked is 1/p which means n/(n-1). [Use above result]

Similarly, the number of trials needed before 3rd new coupon is collected is n/(n-2)

Using Linearity of expectation:

Total number of expected trials $= 1 + n/(n-1) + n/(n-2) + n/(n-3) + ... + n/2 + n/1$
$= n[1/n + 1/(n-1) + 1/(n-2) + 1/(n-3) + ...+ 1/2 + 1/1]$
$= n * H_n$
$= nLogn$

Since Logn <= Hn <= Logn + 1, we need to buy around nLogn lots to collect all n coupons.

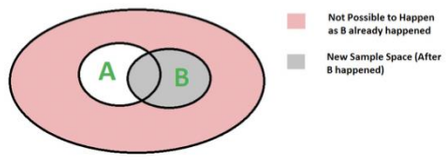
Mathematical Background

Conditional Probability

Conditional probability P(A | B) indicates the probability of happening of event 'A' given that the event B happened.

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

We can easily understand above formula using below diagram. Since B has already happened, the sample space reduces to B.
So the probability of A happening becomes P(A ∩ B) divided by P(B)



Example:
In a batch, there are 80% C programmers, and 40% are Java and C programmers.
A random student was picked and he was c programmer, what is probability that he is also java programmer?

Let A --> Event that a student is Java programmer
B --> Event that a student is C programmer
P(A|B) = P(A ∩ B) / P(B)
= (0.4) / (0.8)
= 0.5
So there are 50% chances that student is also a java programmer.

Bayes' Theorem:

$$P(A \mid B) = \frac{P(B \mid A)}{P(B)} P(A)$$

The formula provides relationship between P(A|B) and P(B|A). It is mainly derived from conditional probability formula.
Consider the below formulas for conditional probabilities P(A|B) and P(B|A):

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \quad ---1$$

$$P(B|A) = \frac{P(B \cap A)}{P(A)} \quad ---2$$

Since P(B ∩ A) = P(A ∩ B), we can replace P(A ∩ B) in first formula with P(B|A)P(A) and get the Bayes formula.

Analyzing Randomized Algorithms

- Some randomized algorithms have deterministic time complexity.
  - Example: Implementation of Karger's algorithm has time complexity as O(E).
  - Such algorithms are called **Monte Carlo Algorithms** and are easier to analyse for worst case.
- On the other hand, time complexity of other randomized algorithms (other than Las Vegas) is dependent on value of random variable.
  - Such Randomized algorithms are called **Las Vegas Algorithms**.
  - These algorithms are typically analysed for expected worst case.
  - To compute expected time taken in worst case, all possible values of the used random variable needs to be considered in worst case and time taken by every possible value needs to be evaluated.
  - Average of all evaluated times is the expected worst case time complexity.
  - Below facts are generally helpful in analysis os such algorithms.
    - Linearity of Expectation
    - Expected Number of Trials until Success.

Classification of Randomized Algorithms

Randomized algorithms are classified in two categories:

- **Las Vegas Algorithms:**
  - These algorithms always produce correct or optimum result.
  - Time complexity of these algorithms is based on a random value and time complexity is evaluated as expected value.
  - ***Example:*** Randomized QuickSort always sorts an input array and expected worst case time complexity of QuickSort is O(nLogn).
- **Monte Carlo Algorithms:**
  - Produce correct or optimum result with some probability.
  - These algorithms have deterministic running time and it is generally easier to find out worst case time complexity.
  - ***Example:*** Implementation of Karger's Algorithm produces minimum cut with probability greater than or equal to $1/n^2$ (n is number of vertices) and has worst case time complexity as O(E).
  - ***Another Example:*** Fermet Method for Primality Testing.

Example to Understand Classification:

Consider a binary array where exactly half elements are 0 and half are 1. The task is to find index of any 1.

- Las Vegas algorithm for this task is to keep picking a random element until we find a 1.
- A Monte Carlo algorithm for the same is to keep picking a random element until we either find 1 or we have tried maximum allowed times say k.

- The Las Vegas algorithm always finds an index of 1, but time complexity is determined as expect value. The expected number of trials before success is 2, therefore expected time complexity is O(1).
- The Monte Carlo Algorithm finds a 1 with probability [1 – (1/2)k]. Time complexity of Monte Carlo is O(k) which is deterministic

Applications of Randomized Algorithms

- Even for sorted array randomized **quick sort** gives O(nlogn) expected time.
- Randomized algorithms have huge applications in **Cryptography**.
- **Load Balancing**
- **Number-Theoretic Applications:** Primality Testing
- **Data Structures:** Hashing, Sorting, Searching, Order Statistics and Computational Geometry.
- **Algebraic identities:** Polynomial and matrix identity verification. Interactive proof systems.
- **Mathematical programming:** Faster algorithms for linear programming, Rounding linear program solutions to integer program solutions
- **Graph algorithms:** Minimum spanning trees, shortest paths, minimum cuts.
- **Counting and enumeration:** Matrix permanent Counting combinatorial structures.
- **Parallel and distributed computing:** Deadlock avoidance distributed consensus.
- **Probabilistic existence proofs:** Show that a combinatorial object arises with non-zero probability among objects drawn from a suitable probability space.
- **Derandomization:** First devise a randomized algorithm then argue that it can be derandomized to yield a deterministic algorithm.

Branch and Bound Approach

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems.
These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.
Branch and Bound solve these problems relatively quickly.

Notes:

- Branch and bound is very useful technique for searching a solution but in worst case, we need to fully calculate the entire tree.
- At best, we only need to fully calculate one path through the tree and prune the rest of it.

Standard Branch and Bound Approach Problems

1. 0/1 Knapsack Problem

Problem:
Given two integer arrays val[0..n-1] and wt[0..n-1] that represent values and weights associated with n items respectively.
Find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to Knapsack capacity W.

Approach-1: Greedy

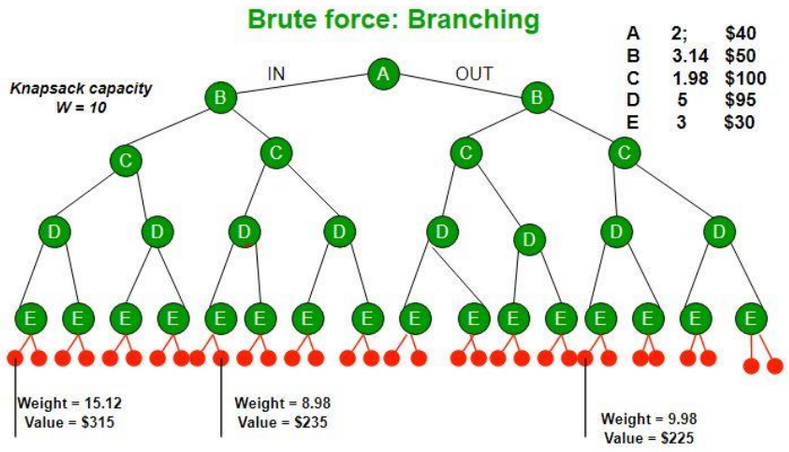
- Pick the items in decreasing order of value per unit weight.
- Works for fractional knapsack problem.
- **May give wrong result for 0/1 knapsack.**

Approach-2: Dynamic Programming (DP)

- In DP, we use a 2D table of size n x W.
- **Doesn’t work if item weights are not integers.**

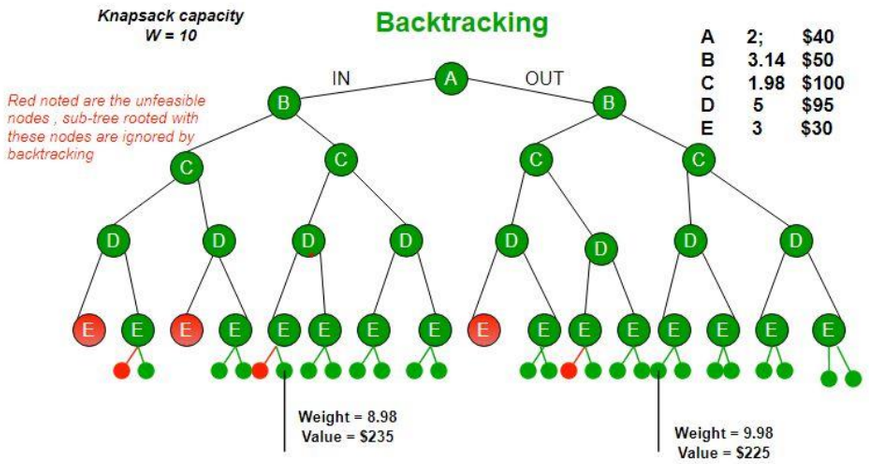
Approach-3: Brute-Force

- With n items, generate all 2^n solutions, check each to see if they satisfy the constraint.
- Save maximum solution that satisfies constraint.
- **Works but exponential complexity.**
- This solution can be expressed as tree.



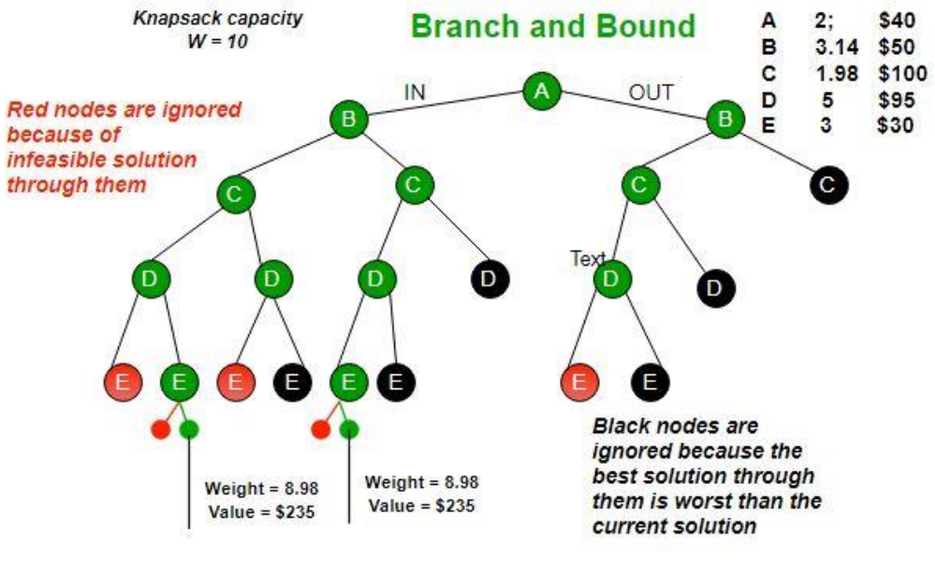
Approach-4: Backtracking

- Use **Backtracking** to optimize the Brute Force solution.
- In the tree representation, do DFS of tree and If when reach a point where a solution no longer is feasible, stop exploring.
- Here, backtracking would be much more effective if we had even more items or a smaller knapsack capacity.
- **Works but time can still be improved**.



Approach-5: Branch and Bound

- The backtracking based solution works better than brute force by ignoring infeasible solutions.
- Can do better (than backtracking) if we know a bound on best possible solution subtree rooted with every node.
- If the best in subtree is worse than current best, we can simply ignore this node and its subtrees.
- So we compute bound (best solution) for every node and compare the bound with current best solution before exploring the node.
- **Works and gives optimal time complexity.**



Finding Bound for every node

- The idea is to use the fact that the Greedy approach provides the best solution for Fractional Knapsack problem.
- To check if a particular node can give a better solution or not, compute optimal solution (through the node) using Greedy approach.
- If the solution computed by Greedy approach itself is less than the best so far, then we can't get a better solution through the node.

Algorithm:

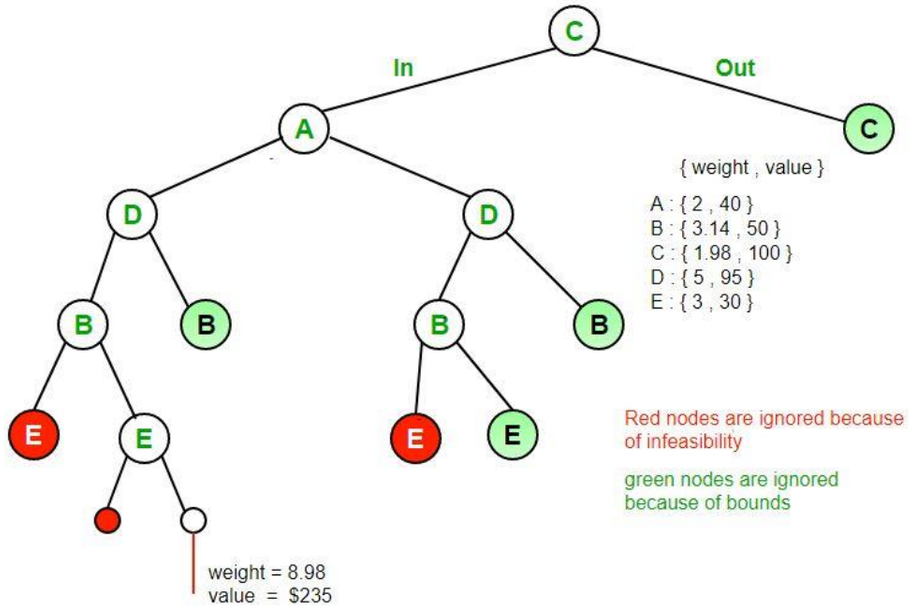
- Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
- Initialize maximum profit, maxProfit = 0
- Create an empty queue, Q.
- Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.
- Do following while Q is not empty.
    - Extract an item from Q. Let the extracted item be u.

- Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
- Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.
- Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.

Illustration:

Input: // First element in every pair is weight of item and second is value of item Item arr[] = [(2, 40), (3.14, 50), (1.98, 100), (5, 95), (3, 30)] Knapsack Capacity W = 10

Output: Maximum possible profit = 235



Implementation:

Output:

Complexity:

- **Time: **
- **Auxilliary Space:**

Miscellaneous Algorithmic Problems

Here are some interesting list of miscellaneous problems.

Problems To Do:

- Find Majority Element in Array