Python Notes Print

Introduction to Algorithm and Python (Monash University)

Time Complexity

The computational (time) complexity of an algorithm is the number of elementary steps T(*n*) needed for computing its output for an input of a size *n*.

How long does a program run? Depends on:

1. the machine used to execute the program;
2. the input (runs longer for larger inputs);
3. the computational complexity of the algorithm.

- **best-case**: this is the complexity of solving the problem for the best input. In our example, the best case would be to search for the value 1. Since this is the first value of the list, it would be found in the first iteration.
- **average-case**: this is the average complexity of solving the problem. This complexity is defined with respect to the distribution of the values in the input data. Maybe this is not the best example but, based on our sample, we could say that the average-case would be when we're searching for some value in the "middle" of the list, for example, the value 2.
- **worst-case**: this is the complexity of solving the problem for the worst input of size n. In our example, the worst-case would be to search for the value 8, which is the last element from the list.

Big-O notation, sometimes called "asymptotic notation", **is a mathematical notation that describes the limiting behavior of a function** when the argument tends towards a particular value or infinity.

Name	Time Complexity
Constant Time	O(1)
Logarithmic Time	O(log n)
Linear Time	O(n)
Quasilinear Time	O(n log n)
Quadratic Time	O(n^2)
Exponential Time	O(2^n)
Factorial Time	O(n!)

Constant Time — O(1)

An algorithm is said to have a constant time when it is not dependent on the input data (*n*). No matter the size of the input data, the running time will always be the same.

Logarithmic Time — O(log n)

An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step (it don't need to look at all values of the input data)

Steps of the binary search:

- Calculate the middle of the list.
- If the searched value is lower than the value in the middle of the list, set a new right bounder.
- If the searched value is higher than the value in the middle of the list, set a new left bounder.
- If the search value is equal to the value in the middle of the list, return the middle (the index).
- Repeat the steps above until the value is found or the left bounder is equal or higher the right bounder.

Linear Time — O(n)

An algorithm is said to have a linear time complexity when the running time increases at most linearly with the size of the input data. This is the best possible time complexity when the algorithm must examine all values in the input data.

Quasilinear Time — O(n log n)

An algorithm is said to have a quasilinear time complexity when each operation in the input data have a logarithm time complexity. It is commonly seen in sorting algorithms.

Quadratic Time — O(n^2)

An algorithm is said to have a quadratic time complexity when it needs to perform a linear time operation for each value in the input data.

Exponential Time — O(2^n)

An algorithm is said to have an exponential time complexity when the growth doubles with each addition to the input data set. This kind of time complexity is usually seen in brute-force algorithms.

Factorial — O(n!)

An algorithm is said to have a factorial time complexity when it grows in a factorial way based on the size of the input data

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

The cheat sheet guide

To recognize the runtime of each problem, I did a small cheat sheet to help you quickly figure out the run-time type and hence, you can improve your algorithm.

- If you are **iterating** over a **single** collection of elements using **one loop**, then run-time will be **O(n)**.
- If you are **iterating** over **half** of the collection, it will be O(n/2) -> **O(n)**.
- If you are **iterating** over **two separate** collections using **two different** loops, so it will become O(n+m) -> **O(n).**
- If you are **iterating** over a **single** collection using **two nested** loops, so it will be **$O(n^2)$**.
- If you are **iterating** over **two different** collections using **two nested** loops, so it will become O(n*m) -> **$O(n^2)$.**
- If you are **sorting** a collection, this becomes **O(n*log(n))**.

Invariant

Loop Invariant Condition:

Loop invariant condition is a condition about the relationship between the variables of our program which is definitely true immediately before and immediately after each iteration of the loop.

Definition: A loop invariant is an assertion inside a loop that is true every time it is reached by the program execution.

Loop *invariants* can be used to analyze behavior of loopy control flows.

Use *assertions* about execution state to reason about programs.

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.

2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Time Complexity: $O(n^2)$ as there are two nested loops.

Explanation: The program selects the first element of an array and switches it when it finds an element smaller to its position and the moves and selects the next number and researches the array again.

```
# Traverse through all array elements
for i in range(len(A)):

    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

    # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]
```

Insertion Sort

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

Time Complexity: O(n*2)

Example:

12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12

11, 12, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13

11, 12, 13, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

```
# Function to do insertion sort
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < arr[j] :
                arr[j + 1] = arr[j]
                j -= 1
        arr[j + 1] = key
```

5, 11, 12, 13, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

5, 6, 11, 12, 13

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

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

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Worst and Average Case Time Complexity: O(n*n). Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted. Not very good performance: O(n2)

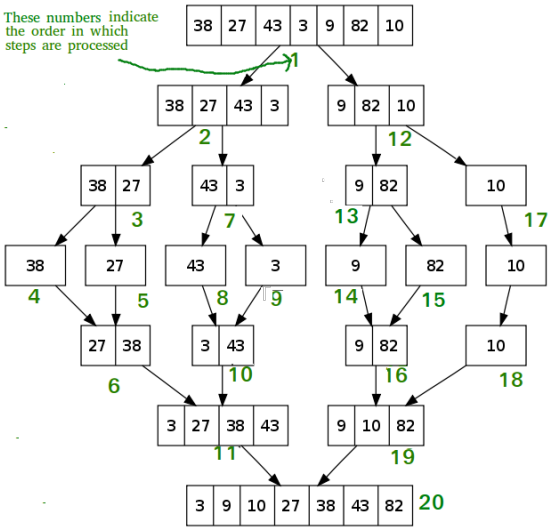
Merge Sort

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.

Divide And Conquer

This technique can be divided into the following three parts:

1. **Divide:** This involves dividing the problem into some sub problem.
2. **Conquer:** Sub problem by calling recursively until sub problem solved.
3. **Combine:** The Sub problem Solved so that we will get find problem solution.



Algorithmic paradigm: **divide-and-conquer**

- halving problem sizes leads to trivial sub problems after logarithmically many reductions
- if not too much overhead: allows to replace linear complexity term by logarithmic term

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \Theta(n)$

Explanation: the array is divide into two until they become pair of two and then compare those pairs and place the smaller number first and then compare the pairs to each other and place the smallest from the pairs

Merge Sort allows worst-case “linearithmic” sorting

Recursion

Recursive functions are a natural way to implement recursive solution strategies (divide-and-conquer, decrease -and-conquer).

A **recursive** function is a function that contains one or more subcalls to itself inside its function body. For termination need subcalls with simpler input until input with no subcalls to self is reached. These inputs are called base case.

Conceptually

Reduce problem to simpler version of itself (cf. decomposition)

decomposition…

…can be thought of from two perspectives:

1. Breaking down programs into sub-programs (components)
2. Breaking down problems into sub-problems

…is most useful if two views coincide, i.e., subprograms correspond to sub-problems

. structures thinking/attention for developing algorithms and reasoning about programs

. leads to re-usable components (because they solve a well-defined problem)

```
def sum(lst):
    n = len(lst)
    if n == 0:
        return 0
    else:
        return lst[0] + sum(lst[1:])
```

sum(lst)=0 if len(lst)==0 (base case)

sum(lst)=lst[0]+lst[1:] if len(lst)>0 (recurrence relation)

Properties of recursive implementation

- One-to-one mapping of sub problems to function calls
- No local variables needed to represent sub problem as part of global problem (calls isolate scope of sub problem)
- No re-assignment/mutation
- Easy to see correctness based on recurrence relation of problem solutions:

sorted(lst)=merged(sorted(lst[:n//2]), sorted(lst[n//2:]))

Decrease and Conquer

Algorithmic paradigm: **decrease-and-conquer**

- decreasing problem size by at least some rate r>1 leads to trivial sub problems after logarithmically many reductions
- if not too much overhead: allows to replace linear complexity term by logarithmic term

Decrease or reduce problem instance to smaller instance of the same problem and extend solution.

Conquer the problem by solving smaller instance of the problem.

Extend solution of smaller instance to obtain solution to original problem .

Basic idea of the decrease-and-conquer technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance. This approach is also known as incremental or inductive approach.

Backtracking

```
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.
3) If all rows have been tried and nothing worked,
   return false to trigger backtracking.
```

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

General approach

- . Find a representation of partial solutions such that all valid option for augmenting a partial solution can be identified as well as when a partial solution is complete
- . Extend problem to find finding all solutions sols(part) that can be reached by a sequence of valid decisions from a specific partial solution part
- . Construct all solutions via partial solutions using recurrence relation: sols(part)=sols(part+[a1])+…+sols(part+[ak])

where [a1,…,ak] are all valid options to augment part

Backtracking Algorithm

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.

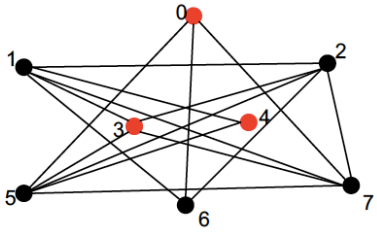
Let's imagine this as an exam task

Task

Write a function `ind_sets(graph, k)` that returns a list of all independent sets of size `k` in the graph `graph` using backtracking and explain your implementation. You can assume access to a function `independent(v, vertices, graph)` that returns if the vertex `v` is not connected to any vertex in `vertices` in the graph `graph` (otherwise).

Questions to answer to apply Backtracking:

1. What are partial solution?
2. What are options to augment partial solution?
3. When is a partial solution complete?



Partial solutions: *ordered* lists of independent vertices

[], [0], [0, 1], ~~[0,1,2]~~, …, ~~[1, 0]~~,…

Python code for ind_sets

```
def ind_sets(graph, k, part=[]):
    if len(part)==k:
        return [part]
    options = []
    for v in range(max(part, default=-1)+1, len(graph)):
        if independent(v, part, graph):
            options += [v]
    res = []
    for o in options:
        res += ind_sets(graph, k, part + [o])
    return res
```

Explanation

Partial solutions are represented by independent sets (in lexicographic order). Complete solutions are then simply partial solutions of length k. Valid options to augment a partial solution part are all vertices v that are greater than the maximal element of a part (to retain lexicographic order) and that are not connected to any element in part (to retain independence set property).

Recall definition of Independent Set

Definition

A subset *X* of vertices of a graph is called an **independent set** if for every pair of vertices *v*, w from *X* there is no edge between *v* and *w*.

Graphs

A **path** is a non-self-intersecting (no two vertices in the sequence can be identical) sequence of vertices such that there is an edge between consecutive vertices in the sequence.
A **cycle** is a path with same start and end vertex.
Graph in which there is a path between any two vertices is called **connected**.

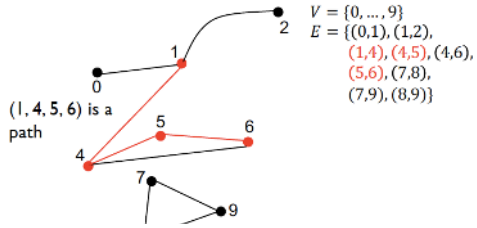


Figure 1 Path

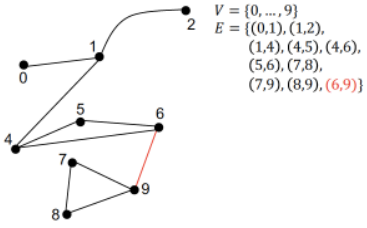


Figure 3 Connection

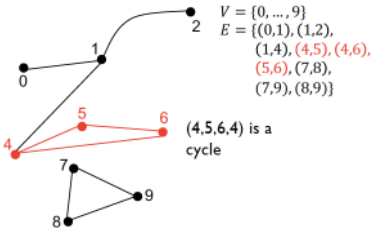
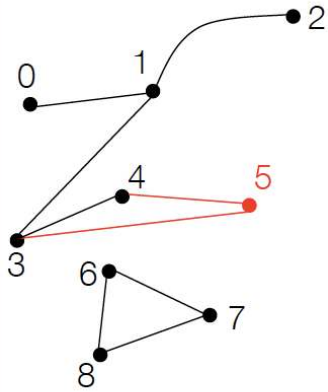


Figure 2 Cycle

Representation as *adjacency matrix*

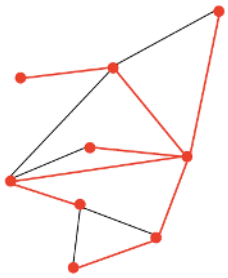
```
graph = \
    [[0,1,0,0,0,0,0,0,0,0],
     [1,0,1,1,0,0,0,0,0,0],
     [0,1,0,0,0,0,0,0,0,0],
     [0,1,0,0,1,1,0,0,0,0],
     [0,0,0,1,0,1,0,0,0,0],
     [0,0,0,1,1,0,0,0,0,0],
     [0,0,0,0,0,0,0,1,1,1],
     [0,0,0,0,0,0,1,0,1,1],
     [0,0,0,0,0,0,1,1,0,0]]

def neighbours(i, g):
    """I: vertex i, graph g
       O: neighbours of i
       For example:
       >>> neighbours(5, graph)
       [3, 4]
    """
    n = len(g)
    res = []
    for j in range(n):
        if g[i][j]==1:
            res.append(j)
    return res
```



Adjacency matrices can be used to represent graphs in Python.

Spanning tree of a graph

Assume $G = (V, E)$ is simple and connected graph.
A ***spanning tree*** of G is a subgraph $T = (V, F)$ of G that
- contains all vertices of G and
- that is a tree (i.e., connected and cycle free)



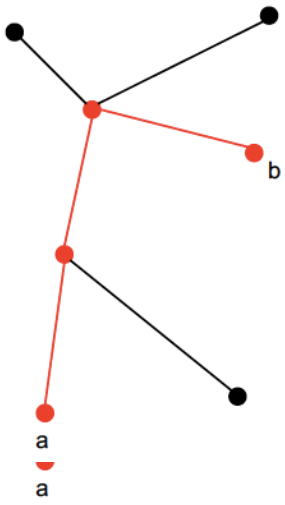


Figure 4 Spanning Tree

Trees
Definition
A simple graph *T* = (*V, E*) is called a **tree** if it is:

- **connected** and
- has no cycles.

Properties
A tree

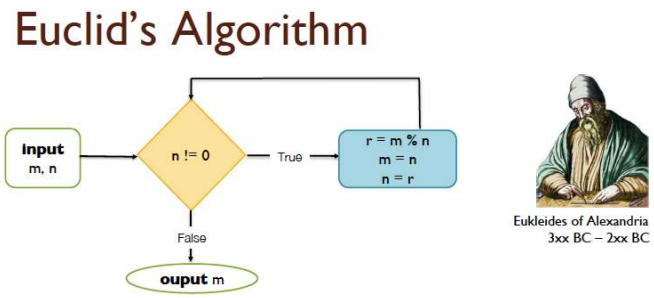
- **is minimally connected**, i.e., removing any edge makes graph disconnected
- contains **unique path** between any two vertices v, w $\in E$

Prim's Minimum Spanning Tree Algorithm
What is Minimum Spanning Tree?
Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A ***minimum spanning tree (MST)*** or **minimum** weight spanning tree for a weighted, connected and undirected graph is a

spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?
A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

How does Prim's Algorithm Work? The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So, the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning* Tree. And they must be connected with the minimum weight edge to make it a *Minimum* Spanning Tree.




```
def gcd(m, n):
    """
    Input : integers m and n such that not n==m==0
    Output: the greatest common divisor of m and n
    """
    while n != 0:
        r = m % n
        m = n
        n = r
    return m
```

Euclid's Algorithm

Binary Search
Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].
A simple approach is to do linear search. The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.
Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.
Trying to find out if a number exists and its position, we divided in into a middle ground and if x is smaller than the middle you check the left hand side and it if is larger, you check the right, whichever side is chosen repeat the same process of getting the middle value, and checking if the middle is greater than or less then or equal to X until you get X. Array as to be sorted.
Binary Search allows logarithmic time look-up of value in sorted sequence.
Time Complexity:
The time complexity of Binary Search can be written as ***T(n) = T(n/2) + c***

```
# It returns location of x in given array arr
# if present, else returns -1
def binarySearch(arr, l, r, x):
    while l <= r:
        mid = l + (r - l) // 2;
        # Check if x is present at mid
        if arr[mid] == x:
            return mid
        # If x is greater, ignore left half
```

```
        elif arr[mid] < x:
            l = mid + 1
        # If x is smaller, ignore right half
        else:
            r = mid - 1
    # If we reach here, then the element
    # was not present
    return -1
```

(Upper) triangular systems

upper triangular, because only coefficients in main diagonal and above are non-zero.

$$\begin{bmatrix}1 & 2 & 1\\0 & 1 & 3\\0 & 0 & 2\end{bmatrix}\begin{bmatrix}a\\b\\c\end{bmatrix}=\begin{bmatrix}2\\7\\6\end{bmatrix} \qquad \begin{bmatrix}1 & 2 & 1\\0 & 1 & 3\\0 & 0 & 2\end{bmatrix}\begin{bmatrix}a\\b\\3\end{bmatrix}=\begin{bmatrix}2\\7\\6\end{bmatrix} \qquad \begin{bmatrix}1 & 2 & 1\\0 & 1 & 3\\0 & 0 & 2\end{bmatrix}\begin{bmatrix}a\\-2\\3\end{bmatrix}=\begin{bmatrix}2\\7\\6\end{bmatrix}$$

$$2c = 6 \qquad b + 3c = 7 \qquad a + 2b + c = 2$$

$$c = 6/2 \qquad b = 7 - 9 \qquad a = 2 + 4 - 3$$

Observations

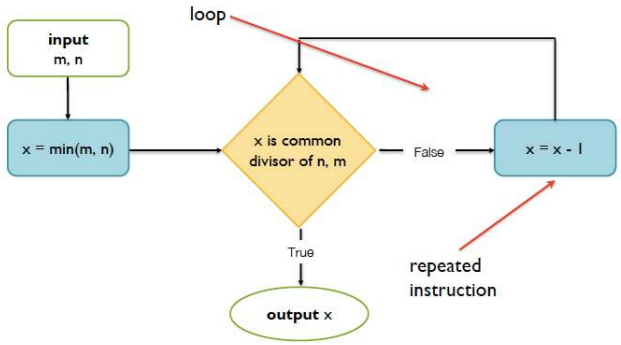
- the greatest possible common divisor is the smaller of the two numbers, e.g. gcd(178, 89) = 89
- the smallest possible divisor is 1, e.g. gcd(97, 53) = 1
- we are after the greatest divisor, e.g. gcd(24, 18) = 6, not 1, 2, or 3

Greatest Common Divisor Problem

Input: two positive integers m and n

Output: greatest common divisor, gcd(m, n)

How to "check every integer"?



"Brute force" Algorithm

check all integers between min(m, n) and 1 (from big to small), output first common divisor encountered

"Brute force" Algorithm check all integers between min(%, ') and 1 (from big to small), output first common divisor encountered

How to check every integer?

Observations

- Depending on input there can be an arbitrary number of integers to check
- Program will always have only a fixed number of instructions

Prim's algorithm finds spanning tree by iteratively adding extension edge to already connected subgraph. Need to **decompose** programs to increase readability and simplify analysis.

Greedy Algorithm to find Minimum number of Coins

Time Complexity: O(N*logN).

Approach: A common intuition would be to take coins with greater value first. This can reduce the total number of coins needed. Start from the largest possible denomination and keep adding denominations while the remaining value is greater than 0.

Algorithms

1. Sort the array of coins in decreasing order.
2. Initialize result as empty.
3. Find the largest denomination that is smaller than current amount.
4. Add found denomination to result. Subtract value of found denomination from amount.
5. If amount becomes 0, then print result.
6. Else repeat steps 3 and 4 for new value of V.

Definition
An **computational problem** is called a decision problem if the required output for each input is Boolean (yes or no).
Inputs for which output is yes (True) are a called yes-input.
Inputs for which output is no (False) are a called no-input.

Theory suggests that NP-complete problems are inherently intractable (unless P=NP)

What are NP, P, NP-complete and NP-Hard problems?
P is set of problems that can be solved by a deterministic Turing machine in **P**olynomial time.
NP is set of decision problems that can be solved by a **N**on-deterministic Turing Machine in **P**olynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time).
Informally, NP is set of decision problems which can be solved by a polynomial time via a "Lucky Algorithm", a magical algorithm that always makes a right guess among the given set of choices (Source Ref 1).