**Big-Ω**: For large values of n the running time of f(n) is at least b⋅g(n)

* + **Ω(f(n))** = if the lim is infinity (Omega)

Diagram

Description automatically generated with medium confidence**Big-Θ**: For large values n the running time of f(n) is at least a⋅g(n)

* + **Θ(f(n)) = if the lim is const(C) (Theta)**

**Big-O**: For large values of n the running time f(n) is at most b⋅g(n)

* + **O(f(n))** = if the lim is trends to 0 (Big-O)

Diagram

Description automatically generated\*n! > 2n

\*(n log n) < 2n

\*n2 > (n log n)

\*5n > n5

\*(log n) < √n

\* This means summation on i where i ranges from 1 to n = nk, where k=2

A picture containing application

Description automatically generated

(n^2 (n+1))/3 ∈ O(n4) : True

(n^2 (n+1))/3 ∈ O(n3) : True

(n^2 (n+1))/3 ∈ Θ(n4) : False

(n^2 (n+1))/3 ∈ Ω(n2) : True

-**Given** an array A of size n, we want to access the ith element in the array, 0<i<n. What will be the time complexity of this operation?

Ans: O(1) because you iterate through one by one until you reach i.

-**Given** an array A of size n, we want to find if an element k belongs to this array. What will be time complexity of this search operation? Assume that we don't know anything about the order of elements in the array.

Ans: O(n), since we do not know the order we operate with however big the array is

-**Given** a sorted array A of size n, we want to find if an element k belongs to this array. What will be the **best time complexity**to perform this search operation? Note: best **time complexity**and not the best time

Ans: O(log n), since it is sorted we don’t work with complexity of O(n), O(1) is too slow….

-**Write** the loop invariant for the following code:

Text

Description automatically generated

Text, letter

Description automatically generatedAns: the loop invariant condition is that 'item' is always maximum among the first i elements of array A.

**Loop invariants**: whatever the holder in the loop is holding throughout the entire loop.

Text

Description automatically generated-What does the algorithm compute? The algorithm returns “true” if its input matrix is symmetric and “false” if it is not.

-What is its basic operation, write the **line number**of code (1, 2, 3, 4, 5, or 6) that is executed maximum number of times? Line 4

-What is the time complexity of this code? O(n2)

Text

Description automatically generated**-Problem**: Given an array of numbers find if there is a subset that adds to a given number. Return True if there exists such subset, else return False. The subset of numbers need not be continuous in the array. We don't know anything about the order of the elements in the array.

Identify which of the strategies can be used to solve this problem: Dynamic programming, Backtracking, Brute force

**Dynamic programming** – the most optimal solution always/substructure. Overlapping subproblems. (is creating an array and using what is already solved (linear F(n)). O(An))

**Back tracking** – gives all possible solutions. Identify the parameters that affect the problem, identify subproblems, recursive formula. Find a solution and then go backward to find other options. Intelligent exhaustive search (exponential time complexity).

Table

Description automatically generated**Divide and conquer –** quick sort, merge sort, binary search (n log n). Break a problem into not overlapping subproblems. Reduces time complexity usually due to recursion.

**Memoization**: In dynamic programming, the technique of storing the previously calculated values

**Greedy**: Hard to design, difficult, efficient compared to dynamic. Best available choice and never looking back. Optimal solution at each step.

**Topological Sort**: Directed acyclic graphs. To order things based on their needs. Used for installing systems that have packages that depend on each other, or job scheduling. Like DFS, O(V+E)

Text, letter

Description automatically generated(Not for undirected graphs with cycles). Solution is not always unique.

Chart, radar chart

Description automatically generated**Prim’s Algorithm**: Picks a random node and builds the spanning tree by using the greedy approach of selecting the edge with the least weight. Naïve: O(VE), Priority queue: O(ElogV). No cycles, can use heap.

Text, letter

Description automatically generatedText, letter

Description automatically generated**Kruskal’s Algorithm**: Uses the greedy approach to find a minimum spanning tree, picks the lowest weighted edge to iterate, if it does not create a cycle then add it to the tree. O(ElogV)

**Dijkstra’s Algorithm:** Used to find the shortest path in a graph that has all non-negative edges. Naïve:O(V2), PriorityQueue:O((V+E)logV)

dist[v] = min{ dist[v] , weight[u,v]+dist[u] }

Graphical user interface, text

Description automatically generated**Greedy Algorithm**: An algorithm that strategically makes the optimal choice at each stage, leading to the optimal solution.

O(n log n)

**NP-Complete proof**: State that A is NP, Find a similar NP-complete algorithm, use that algorithm to solve for A, this proves the solution is correct for all instances.

Table

Description automatically generated

Text, letter

Description automatically generated

Table

Description automatically generatedText

Description automatically generated with medium confidence

Diagram, schematic

Description automatically generated

Text

Description automatically generated**Greedy algorithm quicksort**: sort activities, for I in activities, if curTime ≤ starttime, result.append(i), result +=1, ret result. Time complexity = O(n2)/.

**Question**: The asymptotic runtime of the solution for the combination sum problem is exponential as is all time complexity.

Text, letter

Description automatically generatedUndirected graph: sun of degrees of all edges = 2|E|. Every edge contributes as two to the sum of degrees.

Text

Description automatically generated**Min priority queue**: implements dijkstras

-A spanning tree does not always contain all edges of the graph,

-A graph can have multiple spanning trees.

**Approximation Ratio**= approximation algorithm/optimal solution

**Question**: We use reduction to prove that NP-Completeness of a problem X from A. As a part of reduction we must prove the following statements. Assume A is a NP-Hard problem.

-Statement P: A can be transformed to X in a polynomial time

-Statement Q: We can obtain solution to A from X in polynomial time.

-Removing the max weight from a Hamiltonian cycle results in a spanning tree.

-P=NP: Unknown, For every decision problem there is a polynomial algorithm to solve: False, If a problem can be solved in polynomial time then A is NP: True, If there is a polynomial time reduction from a problem A to circuit SAT then A is NP-hard: False, If problem A is in NP then it is NP-complete: false.

**Bottom-up**: start with base case and build to solution. Start at bottom right corner of array

**Top-down**: solve bigger problem toward base case. Start at top left corner of array.

**BFS**: All eligible nodes from starting point, then all eligible nodes from those points. (queue)

**DFS**: All eligible nodes down one whole branch at a time. (stack)

Dijkstra cannot be negative, because it may not recognize the value correctly.

Prim’s picks a random node and builds a spanning tree from there, while Kruskal starts with the minimal edge available that does not create a cycle and adds that to the tree.

b. Prim’s algorithm adds to what is already there or connected, whereas this is not necessary in the Kruskal’s. Kruskal gauges the tree branches on the cheapest edges available, but not necessarily connected yet.