Amortized Analysis • In Amortize analysis finds the time required to perform a sequence of data structure operations is average of overall operation performed. • It guarantees the average time per operation over worst case performance. • It is used to show the average cost of an operation. There are three methods of amortized analysis 1**) Aggregate Method** • In this method we computes worst case time T(n) for n operations. • Amortized cost is T(n)/n per operation. • It gives the average performance of each operation in the worst case.2) **Accounting Method** • In accounting method we assign different charges to different operations. • The amount we charge in operation is called "amortized cost". • When amortized cost > actual cost then difference is assigned to "Credit". • This credit can be used when amortized cost < actual cost. • The total amortized cost of a sequence of operations must be an upper bound on the total cost of sequence.

Merge sort :Analysis

• Let T(n) be the time taken by this algorithm to sort an array of n elements. • Separating T into U & V takes linear time; merge (U, V, T) also takes linear time. • Now, T(n)=T(n/2)+ T(n/2)+g(n) where g(n) ϵ Θ(n). T(n) = 2t(n/2)+ Θ (n) Applying the general case, l=2, b=2, k=1 Since l = bk the second case applies which yields t(n) ϵ Θ(nlogn). Time complexity of merge sort is Θ (nlogn).

```
Procedure mergesort(T[1,...,n])
    if n is sufficiently small then insert(T)
    else
        array U[1,...,1+n/2],V[1,...,1+n/2]
        U[1,...,n/2] ← T[1,...,n/2]
        V[1,...,n/2] ← T[n/2+1,...,n]
        mergesort(U[1,...,n/2])
        mergesort(V[1,...,n/2])
        merge(U, V, T)
```

Insertion sort:

Best case: Take , j= 1

T(n) = C1n + C2(n-1) + C3(n-1) + C4(n-1) + C5 (n-1)+ c7(n-1) = (C1+C2+C3+C4+C7) n (C2+C3+C4+C7) = an b Thus, T(n) = Θ(n)

Worst case:

Take j =n T (n) = C1n+ C2(n-1)+ C3(n-1)+ C4(∑) +(C5 +C6) ∑ + C7(n-1). = C1n+ C2n+ C3n+ C4 +C5 + C6 + C4 + C5 + C6 + C7n-C2-C3- C7. = n2 (C4 + C5 + C6)+n(C1 +C2 +C3 +C7 + C4 + C5 + C6)-1(C2 +C3 +C7). = an 2 + bn + c. Thus, T(n) = Θ(n2)

Average case: Average case will be same as worst case T(n) = Θ(n2)Time complexity of insertion sort Best case Average case Worst case O(n) O (n2) O (n2)

```
Procedure insert (T[1….n])
    for i ← 2 to n do
        x ← T[i]
        j ← i - 1
        while j > 0 and x < T[j] do
            T[j+1] ← T[j]
            j ← j - 1
        end
    T[j + 1] ← x
end
```

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(logn)	O(logn)
Bubble Sort	O(n)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)
Radix Sort	O(nk)	O(nk)	O(nk)
Tim Sort	O(n)	O(nlogn)	O(nlogn)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)

Quick sort :Analysis

Worst Case • Running time of quick sort depends on whether the partitioning is balanced or unbalanced. • And this in turn depends on which element is chosen as key or pivot element.

The worst case: behavior for quick sort occurs when the partitioning routine produces one sub problem with n-1 elements and one with 0 elements. • In this case recurrence will be, T(n)=T(n-1)+T(0)+Θ(n) T(n)=T(n-1)+ Θ (n) T(n)= Θ (n2) **Best Case:** • Occurs when partition produces sub problems each of size n/2. • Recurrence equation: T(n)=2T(n/2)+ Θ(n) l = 2, b = 2, k = 1, so l = bk T(n)= Θ(nlogn)

Average Case: • Average case running time is much closer to the best case. • If suppose the partitioning algorithm produces a 9-to-1 proportional split the recurrence will be T(n)=T(9n/10)+ T(n/10)+ Θ(n) Solving it, T(n)= Θ(nlogn) • The running time of quick sort is therefore Θ(nlogn) whenever the split has constant proportionality.

Procedure: pivot(T[i,…,j]; var l) {Permutes the elements in array T[i,…,j] and returns a value l such that, at the end, i<=l<=j, T[k]<=p for all i ≤ k < l, T[l]=p, And T[k] > p for all l < k ≤ j, where p is the initial value T[i]}

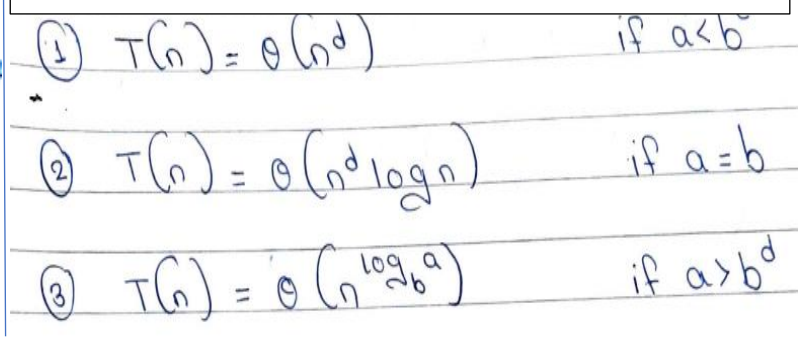
```
P ← T[i]
K ← i; l ← j+1
Repeat k ← k+1 until T[k] > p
Repeat l ← l-1 until T[l] ≤ p
While k < l do
    Swap T[k] and T[l]
    Repeat k ← k+1 until T[k] > p
    Repeat l ← l-1 until T[l] ≤ p
Swap T[i] and T[l]
```

```
Procedure quicksort(T[i,…,j])
{Sorts subarray T[i,…,j] into non decreasing order}
if j – i is sufficiently small then insert (T[i,…,j])
else
 pivot(T[i,…,j],l)
quicksort(T[i,…, l - 1])
quicksort(T[l+1,…,j]
```

dynamic programming • Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to sub problems. • Divide-and-conquer algorithms partition the problem into independent sub problems, solve the sub problems recursively, and then combine their solutions to solve the original problem. • In contrast, dynamic programming is applicable when the sub problems are not independent, that is, when sub problems share sub problems. • In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub problems. • A dynamic-programming algorithm solves every sub problem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the sub problem is encountered. • The development of a dynamic-programming algorithm can be broken into a sequence of four steps. 1. Characterize the structure of an optimal solution. 2. Recursively define the value of an optimal solution. 3. Compute the value of an optimal solution in a bottom-up fashion. 4. Construct an optimal solution from computed information

Principle of optimality • The dynamic programming algorithm obtains the solution using principle of optimality. • The principle of optimality states that “in an optimal sequence of decisions or choices, each subsequence must also be optimal”. • When it is not possible to apply the principle of optimality it is almost impossible to obtain the solution using the dynamic programming approach. • The principle of optimality: “If k is a node on the shortest path from i to j, then the part of the path from i to k, and the part from k to i, must also be optimal.”

Master theorem



Greedy Algorithm works by making the decision that seems most promising at any moment; it never reconsiders this decision, whatever situation may arise later. • Greedy algorithms and the problems that can be solved by greedy algorithms are characterized by most or all of the following features. • To solve a particular problem in an optimal way using greedy approach, there is a set or list of candidates C. • For example: In Make Change Problem - the coins that are available, In Minimum Spanning Tree Problem - the edges of a graph that may be used to build a path, In Job Scheduling Problem - the set of jobs to be scheduled, etc.. • Once a candidate is selected in the solution, it is there forever: once a candidate is excluded from the solution, it is never reconsidered. • To construct the solution in an optimal way, Greedy Algorithm maintains two sets. One set contains candidates that have already been considered and chosen, while the other set contains candidates that have been considered but rejected. The greedy algorithm consists of four functions. 1. Solution Function:- A function that checks whether chosen set of items provides a solution. 2. Feasible Function:- A function that checks the feasibility of a set. 3. Selection Function:- The selection function tells which of the candidates is the most promising. 4. Objective Function:- An objective function, which does not appear explicitly, but

```
Function greedy(C : set): set
{C is the set of candidates.}
S ← Ø {S is a set that will hold the solution}
while C ≠ Ø and not solution(S) do
        x ← select (C)
        C ← C \ {x}
        if feasible ( S U {x}) then
                S ← S U { x}
if solution (S) then return S
else return "no solution found"
```

Depth-First Search of graph • Let G = (N, A) be an undirected graph all of whose nodes we wish to visit. • Suppose it is somehow possible to mark a node to show it has already been visited. • To carry out a depth-first traversal of the graph, choose any node v Є N as the starting point. • Mark this node to show it has been visited. • Next, if there is a node adjacent to v that has not yet been visited, choose this node as a new starting point and call the depth-first search procedure recursively. • On return from the recursive call, if there is another node adjacent to v that has not been visited, choose this node as the next starting point, and call the procedure recursively once again, and so on. • When all the nodes adjacent to v are marked, the search starting at v is finished. • If there remain any nodes of G that have not been visited, choose any one of them as a new starting point, and call the procedure yet again.

```
procedure dfsearch(G)
        for each v Є N do mark[v] ← not-visited
        for each v Є N do
                if mark[v] ≠ visited then dfs(v)
procedure dfs(v)
        {Node v has not previously been visited}
        mark[v] ← visited
        for each node w adjacent to v do
                if mark[w] ≠ visited then dfs(w)
```

```
procedure search(G)
        for each v ε N do mark[v] ← not visited
        for each v ε N do
                if mark[v] ≠ visited then { dfs2 or bfs } (v)
procedure bfs(v)
        Q ← empty-queue
        mark[v] ← visited
        enqueue v into Q
        while Q is not empty do
                u ← first(Q)
                dequeue u from Q
                for each node w adjacent to u do
                        if mark[w] ≠ visited then mark[w] ← visited
                                enqueue w into Q
```

Breadth-First Search of graph • Given a graph G = (V, E) and a distinguished source vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. • It computes the distance (smallest number of edges) from s to each reachable vertex. • It also produces a "breadth-first tree" with root s that contains all reachable vertices. • For any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the smallest number of edges. • The algorithm works on both directed and undirected graphs. • Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. • That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance k + 1.

Back Tracking • In its basic form, backtracking resembles a depth first search in a directed graph. • The graph is usually a tree or at least it does not contain cycles. • The graph exists only implicitly. • The aim of the search is to find solutions to some problem. • We do this by building partial solutions as the search proceeds. • Such partial solutions limit the regions in which a complete solution may be found. • Generally, when the search begins, nothing is known about the solutions to the problem. • Each move along an edge of the implicit graph corresponds to adding a new element to a partial solution. That is to narrow down the remaining possibilities for a complete solution. • The search is successful if, proceeding in this way, a solution can be completely defined. • In this case either algorithm may stop or continue looking for alternate solutions. • On the other hand, the search is unsuccessful if at some stage the partial solution constructed so far cannot be completed.

Minmax Principle • Sometimes it is impossible to complete a search due to large number of nodes for example games like chess. • The only solution is to be content with partial solution. • Minmax is a heuristic approach and used to find move possibly better than all other moves. • Whichever search technique we use, the awkward fact remains that for a game such as chess a complete search of the associated graph is out of the question. • In this situation we have to be content with a partial search around the current position. • This is the principle underlying an important heuristic called Minimax. • Minimax (sometimes Minmax) is a decision rule used in decision theory, game theory, statistics and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario. • Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves, it has also been extended to more complex games and to general decision making in the presence of uncertainty. • A Minimax algorithm is a recursive algorithm for choosing the next move in an n-player game, usually a two-player game.

Algorithm: Traveling-Salesman-Problem
C ({1}, 1) = 0
for s = 2 to n do
for all subsets
S Є {1, 2, 3, … , n} of size s and containing 1
C (S, 1) = ∞
for all j Є S and j ≠ 1
C (S, j) = min {C (S – {j}, i) + d(i, j)
for i Є S and i ≠ j}
Return minj
C ({1, 2, 3, …, n}, j) + d(j, i)

C11:P+S-T+V
C12: R+T
C21: Q+S
C22: P+R-Q+U

Rabin-Karp algorithm is an algorithm used for searching/matching patterns in the text using a hash function. Unlike Naive string matching algorithm, it does not travel through every character in the initial phase rather it filters the characters that do not match and then performs the comparison.A sequence of characters is taken and checked for the possibility of the presence of the required string. If the possibility is found then, character matching is performed.Let us understand the algorithm with the following steps:Let the text be: Text

And the string to be searched in the above text be:Pattern

1. Let us assign a `numerical value(v)/weight` for the characters we will be using in the problem. Here, we have taken first ten alphabets only (i.e. A to J).Text Weights
2. `n` be the length of the pattern and `m` be the length of the text. Here, `m = 10 and n = 3.`

   Let `d` be the number of characters in the input set. Here, we have taken input set {A, B, C, ..., J}. So, `d = 10`. You can assume any suitable value for `d`.
3. Let us calculate the hash value of the pattern.

Algorithm RABIN-KARP-MATCHER(T, P, d, q)

```
n ← length[T];
m ← length[P];
h ← d^(m-1) mod q;
p ← 0;
t0 ← 0;
for i ← 1 to m do
        p ← (dp + P[i]) mod q;
        t0 ← (dt0 + P[i]) mod q
for s ← 0 to n – m do
        if p == ts then
                if P[1..m] == T[s+1..s+m] then
                        print "pattern occurs with shift s"
        if s < n-m then
        ts+1 ← (d(ts – T[s+1]h) + T[s+m+1]) mod q
```

Knuth-Morris and Pratt: introduce a linear time algorithm for the string matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs**1. The Prefix Function (Π):** The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'**2. The KMP Matcher:** With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

```
1  n ← length[T]
2  m ← length[P]
3  π ← COMPUTE-PREFIX-FUNCTION(P)
4  q ← 0                    //Number of characters matched.
5  for i ← 1 to n           //Scan the text from left to right.
6    do while q > 0 and P[q + 1] ≠ T[i]
7         do q ← π[q]   //Next character does not match.
8      if P[q + 1] = T[i]
9        then q ← q + 1     //Next character matches.
10     if q = m             //Is all of P matched?
11       then print "Pattern occurs with shift" i - m
12            q ← π[q]   //Look for the next match.
```

P and NP Problems P Problems • The class P consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in time O(n k) for some constant k, where n is the size of the input to the problem. Fundamental complexity classes. • The class of problems that can be solved in polynomial time is called P class. • These are basically tractable. Few examples of P class problems are, 1. A set of decision problems with yes/no answer. 2. Calculating the greatest common divisor. 3. Sorting of n numbers in ascending or descending order. 4. Searching of an element from the list, etc. NP Problems • NP = Non-Deterministic polynomial time • NP means verifiable in polynomial time • The class NP consists of those problems that are "verifiable" in polynomial time. • What we mean here is that if we were somehow given a "certificate" of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem. • NP is one of the most fundamental classes of problems in computational complexity theory. • The abbreviation NP refers to non-deterministic polynomial time. • These problems can be solved in non-deterministic polynomial time. • For example 1. Knapsack problem O(2n/2) 2. Travelling salesperson problem (O(n22 n)). 3. Graph coloring problem. 4. Hamiltonian circuit

NP hard and NP Complete problems NP Complete • The class of problems "NP-complete stands for the sub-lass od decision problems in NP that are hardest. • The class NP-complete is abbreviated as NPC and comes from: -Non-deterministic polynomial -Complete-"Solve one, solve them all" A decision problem L is said to be NP-Complete if: (i) L is in NP that means any given solution to this problem can be verified quickly in polynomial time. (ii) Every problem is NP reducible to L in polynomial time. • It means that if a solution to this L can be verified in polynomial time then it can be shown to be in NP. • A problem that satisfies second condition is said to be NP-hard that will be examined in recent. • Informally it is believed that if a NPC problem has a solution in polynomial time then all other NPC problems can be solved in polynomial time The list given below is the examples of some well-known problems that are NP-complete when expressed as decision problems. i. Boolean circuit satisfiability problem(C-SAT). ii. N-puzzle problem. iii. Knapsack problem. iv. Hamiltonian path problem. v. Travelling salesman problem. vi. Sub graph isomorphism problem. vii. Subnet sum problem. viii.Clique Decision Problem (CDP). ix. Vertex cover problem. x. Graph coloring problem. • The following techniques can be applied to solve NPC problems and they often give rise to substantially faster algorithms: i. Approximation Approach. ii. Randomization. iii. Restriction. iv. Parameterization. v. Heuristics

NP Hard The class NP-hard written as NPH or NP-H stands for non-deterministic polynomial time hard. The class can be defined as: i. NPH is a class of problems that are "At least as hard as the hardest problems in NP" ii. A problem H is NP-hard if there is a NPC problem L that is polynomial time reducible to H (i.e. L H). iii. A problem H is said to be NPH if satisfiability reduces to H. iv. If a NPC problem L can be solved in polynomial time by an oracle machine with an oracle for H. • NP-hard problems are generally of the following types-decisions problems, search problems, or optimization problems

• To prove a problem H is NP-hard, reduce a known NP-hard problem to H. • If a NPH problem can be solved in polynomial time, then all NPC problems can also be solved in polynomial time. As a result, all NPC problems are NPH, but all NPH problems are not NPC.