MOD 1

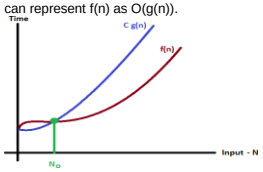
Asymptotic Notations and their properties

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete

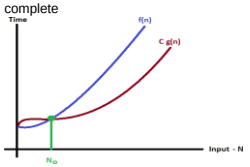
Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n0, C > 0 and n0 >= 1. Then we can represent f(n) as O(g(n)).



Omega Notation, Ω

Consider function **f(n)** the time complexity of an algorithm and **g(n)** is the most significant term. If **f(n) >= C . g(n)** for all **n >= n0**, **C > 0** and **n0 >= 1**. Then we can represent **f(n)** as **Ω(g(n))**.

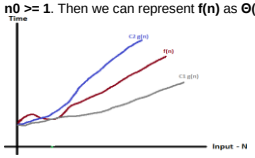
The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete



Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −

Consider function **f(n)** the time complexity of an algorithm and **g(n)** is the most significant term. If **C1 g(n) <= f(n) >= C2 g(n)** for all **n >= n0**, **C1, C2 > 0** and **n0 >= 1**. Then we can represent **f(n)** as **Θ(g(n))**.



MOD 3.2

Knapsack ALGORITHM

```
void GreedyKnapsack (float m, int n)
{
// p[i]/w[i] ≥ p[i+1]/w[i+1]
 for (int i=1; i<=n; i++)
 x[i] = 0.0;
 float u = m;
 for (i=1; i<=n; i++)
 {
 if(w[i] > u) break;
 x[i] = 1.0;
 u = u – w[i];
 }
 if (i<=m) x[i] = u/w[i];
}
```

MOD 1.1

Best, worst and Average Case Complexities

Worst Case – running time of an algorithm is the function defined by maximum number of steps taken on any instance of size n

Outer for loop runs for (n-1) times

In worst case the inner while loop runs for (n-2) times : *10 11 12 13*, Element to insert : 9 1+2+3+...+(n-1) = ((n-1)*n)/2 => **O(n²)**

Average Case - running time of an algorithm is the function defined by an average number of steps taken on any instance of size n.

Outer for loop runs for (n-1) times

On average the inner while loop runs for (n-2)/2 times // Eg - Sorted set : 2 4 *10 12*, Element to insert : 9

Best Case- running time of an algorithm is the function defined by minimum number of steps taken on any instance of size n.

Outer for loop runs for (n-1) times

Inner while loop exits the loop in the very first comparison // Eg - Sorted set : 2 4 5 7, Element to insert : 9

1+1+1+...+(n-1) times => **Ω(n)**

MOD 1.2

INSERTION SORT ALGORITHM

Start with an empty hand with a pack of cards(faced down) on a table.

Pick one card at a time from the table and insert it in the correct position in your hand.

To find the correct position of the card, compare it with each card in the hand from right to left.

(Notice that cards in the hand are already sorted)

Algorithmically :

Insertion Sort(A)

1. for i = 2 to length(A) *// Start with the 2nd element because the first element is trivially sorted*
2. x=A[i] *// x is the element you want to insert in right place into the already sorted set of elements*
3. j=i-1 *// Last index of the already sorted elements because thats where you want to start comparing x*
4. while j>0 and A[j]>x *// Whenever there is an element greater than x*
5. A[j+1]=A[j] *// shift it to the right*
6. j = j-1
7. end while
8. A[j+1]=x *// The correct position to insert x*
9. end for

MOD 2

Applications of Depth First Search

1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) Detecting cycle in a graph

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the

graph and check for back edges. (See this for details)

3) Path Finding

We can specialize the DFS algorithm to find a path between two given vertices u and z

4) Finding Strongly Connected Components of a graph

A directed graph is called strongly connected if there is a path from each vertex in the graph

to every other vertex. (See this for DFS based algo for finding Strongly Connected Components)

5) Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

MOD 5

Randomized algorithms

A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased bits, and it is then allowed to use these random bits to influence its computation. An algorithm is randomized if its outputs is determined by the input as well as the values produced by a random- number generator. Randomized algorithm is one that make use of a randomizer such as a random number generator. Some of the decisions made in the algorithm depend on the output of the randomizer. Since the output of any randomizer might differ in an unpredictable way from run to run, the output of a randomized algorithm could also differ from run to run for the same input so, the execution time of a randomized algorithm could also vary from run to run for the same input.

Two classes:

Las vegas Algorithm always produces the same output of the same input. The execution time of a Las Vegas algorithm depends on the output of the randomizer. if we are lucky athe algorithm might terminate fast and if not might run for alonger period of time . in general its runtime for each input is a random variable whose expectation is bounded.

The second algorithm whose output might differ from run to run for the same input. These are called Monte Carlo algorithm. Consider any problem for which there are only two possible answers, say yes or no. if a Monte Carlo algorithm might give incorrect answer depending on the input of the randomizer. We require that the probability of an incorrect answer from a Monte Carlo Algorithm be low.

MOD 5

RANDOMIZED QUICK SORT ALGORITHM

In randomized quick sort, we will pick randomly an element as the pivot for partitioning. The expected runtime of any input is O(nlogn).

Analysis of randomized Quick sort

Lets s(i) be the ith smallest in the input list S. Xij is a random variable such that Xij-1 ifs (i) is compared with s(j); xij =0 otherwise.

Expected runtime t of randomized Qs is:

$$t = E\left[\sum_{i=1}^{n}\sum_{j\geq 1} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j\geq 1} E[X_{ij}]$$

E[xij] is the expected value of Xij over the set of all random choices of the pivots, which is equal to the probability pij that s(i) will be compared with s(j)

The randomized quick sort algorithm uses i<- random (p,r) to be the new pivot element

Randomized-Partition (A, p, r)

1. $i \leftarrow$ Random (p, r)
2. exchange $A[r] \leftrightarrow A[i]$
3. return Partition (A, p, r)

Randomized-Quicksort (A, p, r)

1. If $p < r$
2. then $q \leftarrow$ Randomized-Partition (A, p, r)
3. Randomized-Quicksort $(A, p, q-1)$
4. Randomized-Quicksort $(A, q+1, r)$

MOD 2

Algorithm: Breadth-First Search Traversal

BFS(V, E, s)

1.foreachuin V−{s}▷for each vertex u in V[G] except s.

2. do color[u]←WHITE
3. d[u]←infinity
4. π[u]←NIL
5. color[s]←GRAY ▷ Source vertex discovered
6. d[s]←0 ▷ initialize
7. π[s]←NIL ▷ initialize
8. Q←{} ▷ Clear queue Q
9. ENQUEUE(Q, s)
10 while Q is non-empty
11. do u ←DEQUEUE(Q) ▷ That is, u = head[Q]
12. for each v adjacent to u ▷ for loop for every node along with Edge.
13. do if color[v]←WHITE
14. then color[v]←GRAY
15. d[v]←d[u] + 1
16. π[v]← u
17. ENQUEUE(Q, v)
18. DEQUEUE(Q)
19. color[u]←BLACK

Analysis

The while-loop in breadth-first search is executed at most |V| times. The reason is thatevery vertex enqueued at most once. So, we have O(V).• The for-loop inside the while-loop is executed at most |E| times if G is a directedgraph or 2|E| times if G is undirected. The reason is that every vertex dequeued atmost once and we examine (u, v) only when u is dequeued. Therefore, every edgeexamined at most once if directed, at most twice if undirected. So, we have O(E).

MOD2.1

Algorithm Depth-First Search

The DFS forms a depth-first forest comprised of more than one depth-first trees. Each tree is made of edges (u, v) such that u is gray and v is white when edge (u, v) is explored. The following pseudocode for DFS uses a global timestamp time**.**

DFS (V, E)

1. for each vertex u in V[G]
2.docolor[u]←WHITE
3. π[u]←NIL
4. time←0
5. for each vertex u in V[G]
6. do if color[u]←WHITE
7. then DFS-Visit(u)

DFS-Visit(u)

1. color[u]←GRAY ▷ discover u
2. time←time + 1
3. d[u]←time
4. for each vertex v adjacent to u ▷ explore (u, v)
5. do if color[v]←WHITE
6. then π[v]← u
7. DFS-Visit(v)
8. color[u]←BLACK
9. time←time + 1
10. f[u]←time

MOD 2.3

STRONGLY CONNECTED COMPONENT

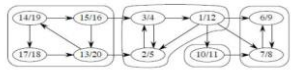
A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph.

Decomposing a directed graph into its strongly connected components is a classic application of depth-first search.

Given digraph or directed graph G = (V, E), a strongly connected component (SCC) of

G is a maximal set of vertices C subset of V, such that for all u, v in C, both u ->v and

v-> u; that is,both u and v are reachable from each other. In other words, two vertices of directed graph arein the same component if and only if they are reachable from each other.



The above directed graph has 4 strongly connected components: C1, C2, C3 and C4. If G has an edge from some vertex in Ci to some vertex in Cj where i ≠ j, then one can reach any vertex in Cj from any vertex in Ci but not return. In the example, one can

reach any vertex inC2 from any vertex in C1 but cannot return to C1 from C2

ALGORITHM

A DFS(G) produces a forest of DFS-trees. Let C be any strongly connected component of G,let v be the first vertex on C discovered by the DFS and let T be the DFStree containing v when DFS-visit(v) is called all vertices in C are reachable from v along paths containing visible vertices; DFS-visit(v) will visit every vertex in C, add it to T as a descendant of v.

STRONGLY-CONNECTED-COMPONENTS (G)

1. Call DFS(G) to compute finishing times f[u] for all u.
2. Compute GT
3. Call DFS(GT

), but in the main loop, consider vertices in order of Decreasing f[u] (as computed in first DFS)

4. Output the vertices in each tree of the depth-first forest formed in second DFSas a separate SCC.

strongly connected component - O(V+E)

Consider a graph G = (V, E)

MOD 2.3

TOPOLOGICAL SORTING

A cycle in a diagraph or directed graph G is a set of edges, {(v1, v2), (v2, v3), ..., (vr−1 , vr)}

where v1 = vr. A diagraph is acyclic if it has no cycles. Such a graph is often referred to as a

directed acyclic graph, or DAG

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

TOPOLOGICAL-SORT(V, E)

1. Call DFS(V, E) to compute finishing times f[v] for all v in V
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

We can perform a topological sort in time ,O(V+E) , since depth-first search takes time and

it takes time to insert each of the Vvertices onto the front of the linked list.

MOD 3

Divide & Conquer

1. The divide-and-conquer paradigm involves three steps at each level of the recursion:
 • Divide the problem into a number of sub problems.
 • Conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straightforward manner.
 • Combine the solutions to the sub problems into the solution for the original problem.
2. They call themselves recursively one or more times to deal with closely related sub problems.
3. D&C does more work on the sub-problems and hence has more time consumption.
4. In D&C the sub problems are independent of each other.
5. Example: Merge Sort, Binary Search

Dynamic Programming

1. The development of a dynamic-programming algorithm can be broken into a sequence of four steps. a. Characterize the structure of an optimal solution.b. Recursively define the value of an optimal solution. c. Compute the value of an optimal solution in a bottom-up fashion.d. Construct an optimal solution from computed information
2. Dynamic Programming is not recursive.
3. DP solves the sub problems only once and then stores it in the table.
4. In DP the sub-problems are not independent.
5. Example: Matrix chain multiplication

MOD 3.1

GREEDY STRATEGY

In a greedy method we attempt to construct an optimal solution in stages. At each stage we make a decision that appears to be the best (under some criterion) at the time. A decision made at one stage is not changed in a later stage, so each decision should assure feasibility. Greedy algorithms do not always yield a genuinely optimal solution. In such cases the greedy method is frequently the basis of a heuristic approach. Even for problems which can be solved exactly by a greedy algorithm, establishing the correctness of the method may be a non-trivial process

In general, greedy algorithms have five pillars:

1. A candidate set, from which a solution is created
2. A selection function, which chooses the best candidate to be added to the solution
3. A feasibility function, that is used to determine if a candidate can be used to contribute to a solution
4. An objective function, which assigns a value to a solution, or a partial solution
5. A solution function, which will indicate when we have discovered a complete solution

Control Abstraction

```
SolutionType Greedy (type a[], int n)
{
 SolutionType solution = EMPTY;

for(int i=1; i<=n; i++)
 {
 Type x = Select(a);
 If Feasible(solution, x)
 solution = Union (solution, x);
} return solution; }
```

Select – select an input from a[] and removes it from the array

Feasible – check feasibility

Union - combines x with solution and updates objective function

MOD 3.3

Kruskal's Algorithm

Kruskal's algorithm is another algorithm that finds a minimum spanning tree for a connected weighted graph. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's Algorithm builds the MST in forest. Initially, each vertex is in its own tree in forest. Then, algorithm considers each edge in turn, order by increasing weight. If an edge (u, v) connects two different trees, then (u, v) is added to the set of edges of the MST, and two trees connected by an edge (u, v) are merged into a single tree on the other hand, if an edge (u, v) connects two vertices in the same tree, then edge (u, v) is discarded. The resultant may not be a tree in all stages. But can be completed into a tree at the end.

ALGORITHM

```
Float kruskal (int E[][], float cost[][], int n, int t[][2])
{
int parent[w];
 consider heap out of edge cost;
 for (i=1; i<=n; i++)
 parent[i] = -1; //Each vertex in different set
 i=0;
 mincost = 0;
 while((i<n-1) && (heap not empty))
 {
 Delete a minimum cost edge (u,v) from the heap and reheapify;
 j = Find(u); k = Find(v); // Find the set
 if (j != k)
 {
 i++;
 t[i][1] = u;
 t[i][2] = v;
 mincost += cost[u][v];
 Union(j, k);
 }
 if (i != n-1) printf(—No spanning tree \nll);
 else return(mincost);
 }
}
```

MOD 2

Applications of Breadth First Traversal

1) Social Networking Websites: In social networks, we can find people within a given distance ‗k' from a person using Breadth First Search till ‗k' levels.
2) GPS Navigation systems: Breadth First Search is used to find all neighboring locations.
3) Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
4) In Garbage Collection: Breadth First Search is used in copying garbage collection using Cheney's algorithm. Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:
5) Cycle detection in undirected graph: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.

MOD 2

Dijkstra's Algorithm

Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

Algorithm Steps:

*Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
*Initialize parent of each node to be NIL.
*Push the source vertex in a min-priority queue as the comparison in the min-priority queue will be according to vertices distances.
*Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
*Update the distances of the connected vertices to the popped vertex in case of "v.d >u.d + w(u,v)", then push the vertex with the new distance to the priority queue.
*If the popped vertex is visited before, just continue without using it.
*Apply the same algorithm again until the priority queue is empty

MOD 2

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.
Step 2: After insertion, check the Balance Factor of every node.
Step 3: If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.
Step 4:If theBalance Factorof any node is other than0 or 1 or -1then tree is said to be imbalanced. Then perform the suitable Rotation to make it balanced. And go for next operation.

MOD 4

Branch and bound is another algorithm technique that we are going to present in our multi-part article series covering algorithm design patterns and techniques. B&B, as it is often abbreviated, is one of the most complex techniques and surely cannot be discussed in its entirety in a single article. Thus, we are going to focus on the so-called A* algorithm that is the most distinctive B&B graph search algorithm.

If you have followed this article series then you know that we have already covered the most important techniques such as backtracking, the greedy strategy, divide and conquer, dynamic programming, and even genetic programming. As a result, in this part we will compare branch and bound with the previously mentioned techniques as well. It is really useful to understand the differences.

MOD 4

BACK TRACKING

Backtracking is a type of algorithm that is a refinement of brute force search. In backtracking, multiple solutions can be eliminated without being explicitly examined, by using specific properties of the problem. It systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors (x1, ..., xn) of values and by traversing, in a depth first manner, the domains of the vectors until the solutions are found. When invoked, the algorithm starts with an empty vector. At each stage it extends the partial vector with a new value. Upon reaching a partial vector (x1, ..., xi) which can't represent a partial solution, the algorithm backtracks by removing the trailing value from the vector, and then proceeds by trying to extend the vector with alternative values.

Control Abstraction

```
Void Backtrack (int k)
{
// While entering assume that first k-1 values x[1], x[2],….,x[k-1] of the
solution
// vector x[1:n] have been assigned.
for (each x[k] such that x[k] ∈ T(x[1],…,x[k-1])) do
{
if (Bk(x[1],…,x[k-1]!=0)) then
{
if ((x[1],…,x[k]) is a path to an answer node)
write x[1:k];
if (k<n) Backtrack(k+1);
}}}
```

Back track approach

– Requires less than m trials to determine the solution
– Form a solution (partial vector) one component at a time, and check at every step if this has any chance of success
– If the solution at any point seems not-promising, ignore it
– If the partial vector (x1, x2, . . . , xi) does not yield an optimal solution, ignore mi+1···mn possible test vectors even without looking at them.
– Effectively, find solutions to a problem that incrementally builds candidates to the solutions, and abandons each partial candidate that cannot possibly be completed to a valid solution.

MOD 4

N Queens Problem

```
Algorithm Nqueens (k, n)
// using backtracking, this procedure prints all
// possible placements of n queens on an n x n
// chessboard so that they are nonattacking.
{
 for í: =1 to n do
 {
 if Place (k, í) then
 {
 x [k] : = í;
 if (k=n) then write (x[1:n]);
 else Nqueens (k+1, n);
 } } }
```

MOD 4

The Traveling Salesman problem

A salesman spends his time in visiting cities cyclically. In his tour he visits each city just once, and finishes up where he started. In which order should he visit to minimize the distance traveled.

The problem an be represented using a graph. Let G=(V,E) be a directed graph with cost cij, cij >0 for all <i j> E and cij = ∞ for all <j,t> E. |V| = n and n>1. We know that tour of a graph includes all vertices in V and cost of tour is the sum of the cost of all edges on the tour. Hence the traveling salesman problem is to minimize the cost.

Application

The traveling salesman problem can be correlated to many problems that we find in the day to day life. For example, consider a production environment with many commodities manufactured by same set of machines. Manufacturing occur in cycles. In each production cycle n different commodities are produced. When machine changes from product i to product j, a cost Cij is incurred. Since products are manufactured cyclically, for the change from last commodity to the first a cost is incurred. The problem is to find the optimal sequence to manufacture the products so that the production cost is minimum.

MOD 5

Approximation algorithms

Approximation algorithms are algorithms designed to solve problems that are not solvable in polynomial time for approximate solutions. These problems are known as NP complete problems. These problems are significantly effective to solve real world problems, therefore, it becomes important to solve them using a different approach.

NP complete problems can still be solved in three cases: the input could be so small that the execution time is reduced, some problems can still be classified into problems that can be solved in polynomial time, or use approximation algorithms to find near-optima solutions for the problems.

Performance Ratios

The main idea behind calculating the performance ratio of an approximation algorithm, which is also called as an approximation ratio, is to find how close the approximate solution is to the optimal solution.

The approximate ratio is represented using ρ(n) where n is the input size of the algorithm, C is the near-optimal solution obtained by the algorithm, C* is the optimal solution for the problem.

The algorithm has an approximate ratio of ρ(n) if and only if –

$$max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \leq \rho(n)$$

The algorithm is then called a ρ(n)-approximation algorithm.

Mod 5

P CLASS

The P in the P class stands for Polynomial Time. It is the collection of decision problems(problems with a "yes" or "no" answer) that can be solved by a deterministic machine in polynomial time.

Features:

The solution to P problems is easy to find.

P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

This class contains many natural problems:

1)Calculating the greatest common divisor.

2)Finding a maximum matching.

3)Decision versions of linear programming

NP CLASS

Features:

The solution to P problems is easy to find. P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

This class contains many natural problems:

1)Calculating the greatest common divisor.

2)Finding a maximum matching.

3)Decision versions of linear programming

NP COMPLETE

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.

Features:

NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.

If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

Some example problems include:

1)Decision version of 0/1 Knapsack.

2)Hamiltonian Cycle**.**

3)Satisfiability.

4)Vertex cover.

NP HARD

An NP-hard problem is at least as hard as the hardest problem in NP and it is a class of problems such that every problem in NP reduces to NP-hard.

Features:

All NP-hard problems are not in NP.

It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.

A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

Some of the examples of problems in Np-hard are:

Halting problem.

Qualified Boolean formulas.

No Hamiltonian cycle.

MOD 5

BIN PACKING PROBLEM

Given n items of different weights and bins each of capacity c. Assign each item to a bin such that number of total used bins is minimized. It may be assumed that all items have weights smaller than bin capacity.

• Lower Bound

The lower bound is the minimum number of bins required

Min no. of bins >= Ceil ((Total Weight) / (Bin Capacity))

Example 1)

Weight of the items = {5,4,7,1,3}

bin capacity = 1

lower bound = minimum number of bins required = Ceil ((Total Weight)/(Bin Capacity))

= Ceil((5+4+7+1+3)/10)= Ceil (20/10) = 2

ie, 2 bins.

1. Next fit:

When processing next item, check if it fits in the same bin as the last item. Use a new bin only if it does not.

2. First Fit:

• When processing the next item, scan the previous bins in order and place the item in the first bin that fits. Start a new bin only if it does not fit in any of the existing bins

3. Best Fit:

• The idea is to places the next item in the tightest spot. That is, put it in the bin so that the smallest empty space is left.

4. First Fit Decreasing:

• We first sort the array of items in decreasing size by weight and apply first-fit algorithm

5. Best fit decreasing algorithm

• We first sort the array of items in decreasing size by weight and apply Best-fit algorithm

MOD 5

GRAPH COLORING

Graph coloring refers to the problem of coloring vertices of a graph in such a way that no two adjacent vertices have the same color.

• This is also called the vertex coloring problem.

• If coloring is done using at most k colors, it is called k-coloring.

• The smallest number of colors required for coloring graph is called its chromatic number. .

Graph coloring problem is both, decision problem as well as an optimization problem.

• A decision problem is stated as, "With given M colors and graph G, whether such color scheme is possible or not?"

. The optimization problem is stated as, "Given M colors and graph G, find the minimum number of colors required

for graph coloring."

The other graph coloring problems are:

• Edge coloring assigns a color to each edge so that no two adjacent edges share the same color

Face coloring of a planar graph assigns a color to each face or region so that no two faces that share a boundary have

the same color

Applications of Graph Coloring Problem

• Design a timetable

. Sudoku

Register allocation in the compiler

Map coloring

Mobile radio frequency assignment

This problem can be solved using backtracking algorithms as follows:

• List down all the vertices and colors

• Assign color 1 to vertex 1

• If vertex 2 is not adjacent to vertex 1 then assign the same color, otherwise assign color 2.

• Repeat the process until all vertices are colored