UNIT 3: Greedy Algorithm

* INTRODUCTION:-

- Greedy algorithm derives solution step-by-step by looking at information available at this moment and not looking at future prospects.
- All decisions are locally optimal.
- Each decision should be
  1) Feasible: Choice satisfies problem constraints.
  2) Locally optimal: Current best solution should be choosed.
  3) Irrevocable: Once a choice is made it cannot be altered.
- Greedy algos are based on five pillars.
  1) Candidate set: Pool of choices from which solution is constructed; e.g.: Nodes, denominations etc.
  2) Selection function: This function choose best local optimal solution based on some heuristics depending on maximization or minimization problem.

3) Feasibility Function :- This tells if candidate selected by solution selection function is a part of solution set.

4) Objective Function :- It assigns benefits or cost to a partial solution.

5) Solution Function :- Indicates whether complete solution is found or not.

- Control Abstraction :-

```
Algorithm GREEDY_APPROACH(L, n)
// Description : Solve given problem using greedy
                 approach.
// Input : L : List of choices , n : size of solution
// Output : Set Solution

Solution ← ϕ
for i ← 1 to n do
    Choice ← Select (L)
    if (feasible (Choice ∪ Solution)) then
        Solution ← Choice ∪ Solution
    end
end
return Solution
```

- Characteristics :-

1) Greedy Choice Property :- Global optimum solution is derived by making locally optimum choices.

2) Optimal substructure :- Given problem exhibits optimal substructure if the optimal solution to the given problem contains the optimal solution to its sub-problem too.

* Knapsack Problem:
- Binary or 0/1 Knapsack:-
- Item cannot be broken down into parts.
- Problem statement:-
Given set of items having some value or profit and weight associated with it, we need to find set of items such that the total weight of the selected items is less than or equal to given limit and the total value or profit is as large as possible.
- Mathematically:-
Let $X = \langle x_1, x_2, \ldots x_n \rangle$ represent set of items
Let n be the total number of items.
Let $W = \langle w_1, w_2, \ldots w_n \rangle$ and $V = \langle v_1, v_2, \ldots v_n \rangle$ be the weight and value associated with each object respectively.
We can represent the problem as
Maximize $\sum_{i=1}^{n} x_i v_i$ subject to $\sum_{i=1}^{n} x_i w_i \leq M$

where M is the total capacity of knapsack.

Also,
$x_i \in \{0, 1\}$ for binary knapsack
$x_i \in [0, 1]$ for fractional knapsack

- Algorithm
Algorithm BINARY_KNAPSACK(W, V, M)
// W is an array of weights of items
// M is the capacity of knapsack
// V is an array of value of items
// Items are pre-sorted in descending order of v_i/w_i
S ← 0 // Store weights of items considered
i ← 1 // Index variable

```
P ← 0 // Variable to track profit
while S < M do
    if (S + W[i]) ≤ M then
        S ← S + W[i]
        P ← P + V[i]
    end
    i ← i + 1
end
```

- Time complexity:
  If items are pre-sorted :- O(n)
  if items need to be sort :- O(n) + O(nlogn) =
  O(nlogn)

- Fractional Knapsack :-
  - Breaking of items is allowed i.e items can be taken in fractions.
  - Same mathematical representation as binary Knapsack.
- Algorithm :-

```
Algorithm FRACTIONAL_KNAPSACK(X, V, W, M)
// Description : Solve fractional knapsack
                 with greedy approach
// Input : X : Array of n items
           V : Array of value associated with each item
           W : Array of weights of each item.
           M : Capacity of knapsack
// Output : SW : Weight of selected items
            SP : Profit of selected items
// Items are presorted in descending order
of Vi/Wi ratio.
SW ← 0   // Weight of selected items
SP ← 0   // Profit of selected items
```

```
i ← 1
while i ≤ n do
    if (sw + w[i]) ≤ M do
        sp ← sp + v[i]
        sw ← sw + w[i]
    else
        frac = (M - s[w])/w[i]
        sp ← sp + v[i] * frac
        sw ← sw + w[i] * frac
    end
    i ← i + 1
end
```

- Time complexity:-
  Exhaustive approach :- $O(2^n)$
  Sorted approach :- $O(n \log n)$

* Scheduling Algorithms :-

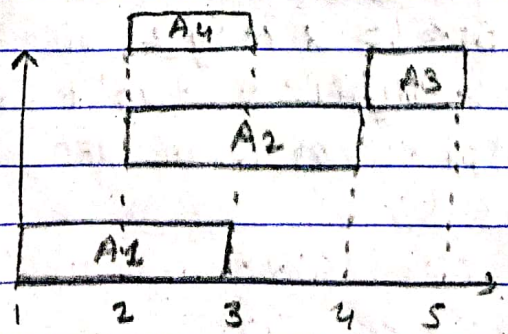
- Job Scheduling :-
  Problem :- Schedule n jobs out of a set of N jobs on a single processor which maximizes profit as much as possible.
- Schedule S is an array of slots s(t), $s(t) \in \{1, 2, 3, \ldots, N\}$ for each $t \in \{1, 2, 3, \ldots, N\}$
- Schedule S is feasible if,
  1) $s(t) = i$ then $t \leq d_i$
  2) Each job can be scheduled at max once.
- Our goal is to find feasible solution S which maximizes profit of scheduled jobs.
- Algorithm:

```
Algorithm JOB_SCHEDULING (J, D, P)
// J is array of N Jobs
// D is deadline of each job
```

```
// P is profit of each job
//sort all jobs in J in decreasing order of profit
S ← ∅  // Set of scheduled jobs
SP ← 0  // Profit of jobs scheduled
for i ← 1 to N do
    if Job[i] is feasible then
        SP ← SP + P[i]
        S ← S ∪ J[i]
    else
        Do not schedule job J[i]
    end
end
```

- Time complexity :-
On ~~worst~~ worst-case N jobs may search for N slots hence TC = O(n²).
if we use set data structure then TC = O(n)

* Activity Selection problem :-
- Problem :- Schedule maximum number of activities that need exclusive access to a resource.
- Span of activity is defined by its starting time and finishing time we have n such activities.



Consider activities = { A1, A2, A3, A4 }
{ 3, 4, 5, 3 } = d
{ A1, A4, A2, A3 }

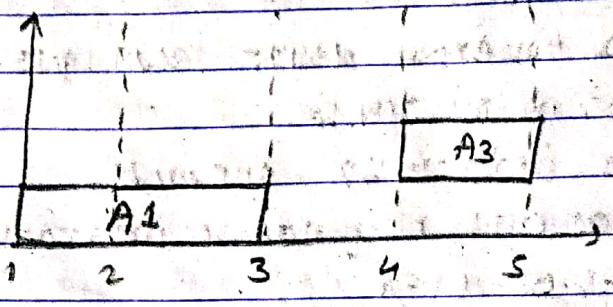
having start time = {1, 2, 4, 2} and finishing time = {3, 4, 5, 3}

- We must select activities such that maximum number of them are completed and their time will not overlap.
- Sort the Activites by their finishing time

F = {3, 3, 4, 5}

S = {1, 2, 2, 4}

A = {A1, A4, A2, A3}



A1 and A3 can be performed optimally.

- Algorithm:

```
Algorithm ACTIVITY_SELECTION(A, s)
// A is set of activities sorted by finishing time
// S is solution set
S ← { A[1] }
i ← 1
j ← 2
for j ≤ n do
    if fi ≤ sj then
        S ← S ∪ A[j]
        i ← j
    end
j ← j + 1
end
```

- Time complexity :-
For each activity at worst there will be (n-1) comparisions hence $O(n^2)$
After sorting single scan will do the work hence TC = $O(n \log n)$

UNIT 3 : Chapter 4
Dynamic Programming

* Introduction :-
- DP is a powerful design technique for many optimization problems.
- It uses bottom-up approach.
- Sub-problems in dynamic programming are not independent.
- DP solves smallest possible problem and uses its solution to solve bigger problems.

* Control Abstraction :-

```
Algorithm DYNAMIC_PROGRAMMING(P)
If solved(P) then
    return lookup(P)
else
    Ans ← SOLVE(P)
    store(P, Ans)
end

Function SOLVE(P)
  if sufficiently_small(P) then
        solution(P)
  else
    Ans1 ← DYNAMIC_PROGRAMMING(P1)
    Ans2 ← DP(P2)
      :
    Ansn ← DP(Pn)
```

return (combine ($P_1, P_2, P_3, \ldots, P_n$))
end

* ~~Characteristics~~ :-
- Principle of Optimality :-
  In an optimal sequence of decisions or choice, each subsequence must also be optimal

* Binomial coefficients :-
- These problem has overlapping subproblems property, we find solution to the same problem again and again.
- We can simply store the solution to the subproblems in a table and refer to it as needed.
- Optimal substructure for binomial coefficient using dynamic programming can be given as :-

$$C[i,j] = \begin{cases} 1, & \text{if } i=j \text{ or } j=0 \\ C[i-1,j-1] + C[i-1,j], & \text{otherwise} \end{cases}$$

- Previously ~~stored~~ solved :- $C[i-1,j-1]$ & $C[i-1,j]$ are used to solve $C[i,j]$
- Mr. Pascal used the same idea to create the Pascal triangle.

- Algorithm :-

```
Algorithm BINOMIAL_COEFF(n, k)
// n is total number of items
// k is number of items to be selected
// C is a 2D Matrix or table
for i ← 0 to n do
    for j ← 0 to k do
```

```
        if i == j or j == 0 then
            C[i,j] = 1
        else
            C[i,j] = C[i-1,j-1] + C[i-1,j]
        end
    end
end
return C[n,k]
```

- Complexity analysis
- We are simply filling the table of size $n \times k$ and returning the last cell value i.e $C[n,k]$ Hence the complexity will be $O(n \cdot k)$ the space complexity will also be the same.

* Optimal Binary Search Tree :- (OBST)
- It is similar to BST, the only catch is, we arrange the nodes according to the probability of searching them frequently i.e the order is maintained but nodes that have high ~~cost frequency or high~~ probability of being searched are closer to the root.
- Consider a dictionary example, a binary search tree will arrange the words in lexicographical order.
- If we manage to keep more frequent words nearer to the root such that they cannot be made more closer to the root we have created an OBST.

- Mathematical foundation :-
  - Any subtree in OBST has keys $k_i \dots k_j$ where $1 \le i \le j \le n$. Sub-tree containing keys $k_i \dots k_j$ have dummy nodes $d_{i-1} \dots d_j$.
  - Consider k_r is root of subtree containing keys $k_i \dots k_j$.
  - Left subtree will contain keys $k_i \dots k_{r-1}$
  - Right subtree will contain keys $k_{r+1} \dots k_j$
  - Recursively OBST algorithm is applied to both right and left subtree of k_r.
  - Let $e[i,j]$ represent cost of searching OBST with keys $k_i \dots k_j$. With n keys our aim is to minimize $e[1,n]$.
  - Base occurs when $j = i-1$
    $\therefore e[i,j] = e[i,i-1] = q_{i-1}$
    where q is probability of unsuccessful search.
  - With k_r as root key and subtree $k_i \dots k_j$ the sum of probability is defined as

$$w(i,j) = \sum_{m=i}^{j} p_m + \sum_{m=i-1}^{j} q_m$$

  - Expected cost can be given as..

$$e[i,j] = p_r + (e[i,r-1] + w[i,r-1] + e[r+1,j] + w[r+1,j])$$

But $w[i,j] = w[i,r-1] + w[r+1,j]$

$$e[i,j] = p_r + (e[i,r-1] + e[r+1,j] + w[i,j])$$

  - Hence recursive formula is

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ \min\{e[i,r-1] + e[r+1,j] + w[i,j]\} \; i \le r \le j & \text{if } i \le j \end{cases}$$

- Algorithm

* Algorithm OBST(p, q, n)
// e[1....n+1, 0...n] : optimal sub tree
// w[1...n+1, 0...n] : Sum of probability
// root // r[1...n, 1...n] : Used to construct OBST
for i ← 1 to n+1 do
    e[i, i-1] ← q_{i-1}
    w[i, i-1] ← q_{i-1}
end
for m ← 1 to n do
    for i ← 1 to n-m+1 do
        j ← i+m-1
        e[i, j] ← ∞
        w[i, j] ← w[i, j-1] + p_j + q_j
        for r ← i to j do
            t ← e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j] then
                e[i, j] = t
                root[i, j] = r
            end
        end
    end
end
return (e, root)

• Complexity :-

$$T(n) = \sum_{m=1}^{n} \sum_{i=1}^{n-m+1} \sum_{j=i}^{n-1+1} \Theta(1)$$

$$= \sum_{m=1}^{n} n^2$$

$$= \Theta(n^3)$$

Do Numericals on DP.

Read this from grooking algorithm.

* 0/1 Knapsack problem
- Same theory as seen in Greedy.

- Mathematical :-

Maximize $\sum_{i=1}^{n} x_i v_i$ Subject to $\sum_{i=1}^{n} x_i w_i \leq M$

x_i : Item, v_i : Profit, w_i : weight of Item, M : knapsack capacity

Now we have two choices while filling knapsack

1) If weight of item is larger than knapsack skip it
   i.e $v[i,j] = v[i-1, j]$ if $j < w_i$
2) If we choose the current item see what choice it beneficial :-
   i) Skip the current item
      $v[i,j] = v[i-1, j]$
   ii) Value of current item + maximum profit that can be gained with remaining weight i.e
      $v[i,j] = v_i + v[i-1, j-w_i]$

- Hence we formulate our problem as

$$v[i,j] = \begin{cases} v[i-1,j] & \text{if } j < w_i \\ \max\{ v[i-1,j], \; v_i + v[i-1, j-w_i] \} & \text{if } j \geq w_i \end{cases}$$

- Algorithm

Algorithm BINARY_KNAPSACK(v, w, M)
// w : weight of each item
// v : value of each
// M : capacity of knapsack

```
for i ← 1 to n do
    V[i, 0] ← 0
end
for i ← 1 to n do
    for j ← 0 to M do
        if w[i] ≤ j then
            V[i, j] = max(V[i-1, j], V[i] +
                          V[i-1, j - w[i]])
        else
            V[i, j] = V[i-1, j]
        end
    end
end
```

- Complexity analysis:
  O(nM) :- n :- items M :- capacity of knapsack
- There is another approach to solve knapsack using DP when M is very large but seems unnecessary for exams.

* Chain Matrix Multiplication:-
- Problem:- In what order, n matrices $A_1, A_2, A_3 \ldots A_n$ should be multiplied to minimize computations to derive result.

- Mathematical representation:-
  Consider three matrices with dimensions as
  $A_1 = 5 \times 4$, $A_2 = 4 \times 6$, $A_3 = 6 \times 2$
  Now, $(A_1 A_2) \cdot A_3 = 180$
  $A_1 \cdot (A_2 A_3) = 88$
  Hence we must simply paranthesize them correctly in order to achieve min. computation

- Let p(n) be alternative parenthesization of n matrices.
- when $n = 1$ $p(n) = 1$ there is only 1 way.
  when $n \geq 2$ $p(n) = \sum_{k=1}^{n-1} p(k)\,p(n-k)$

  p(k) :- Product of first k matrices
  p(n-k) :- Product of remaining matrices.
- We define m[i,j] recursively as follows.
  If i == j then m[i,j] = m[i,i] = 0 as no solution is required since there is only one matrix.
- If i < j, consider optimal parenthesization split matrix between Ak and Ak+1 where product was Ai Ai+1 ... Aj and $i \leq k < j$
  Then m[i,j] is equal to minimum cost of computing sub products Ai...k and Ak+1...j plus cost of multiplying these matrices together

  Optimal substructure

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + d_{i-1} \times d_k \times d_j \} & \text{if } i < j \end{cases}$$

- Algorithm

```
Algorithm MCO(p, i, j)
// p is sequence of matrices
if i == j then
    return 0
m[i,j] = ∞
for j ← i to j − 1 do
    x ← MCO(p, i, k) + MCO(p, k+1, j) +
        p_{i-1} * p_k * p_j
```

```
        if x < m[i,j] then
            m[i,j] = x
        end
    end
end
return m[i,j]
```

Time complexity :-
Iterative without memo :- $O(n^3)$
Recursive without memo :- $O(2^{n-1})$

UNIT 4 :
Chap 5 : Backtracking

* Principle :-
- Backtracking is a recursive approach and a refinement of brute-force method.
- Brute-force evaluates all possible candidates whereas backtracking limits search by eliminating the candidates that do not satisfy certain constraints.
- Backtracking algos are used when we have a set of ~~partial sol^n~~ choices and we do not know which will lead to correct choice.
- Each choice leads to a partial solution, these solutions are explored in DFS order.
- If a partial solution does not satisfy given constraints it will not be explored further.
- This processing can be represented using state space tree.

- Control Abstraction :-

* Control Abstraction:
- In backtracking a solution is defined as n-tuple $\xi X = (x_1, x_2, \ldots, x_n)$ where $x_i = 0$ or $x_i = 1$.
- x_i is choosen from a finite set of components S_i.
- If component x_i is selected then set it as 1 or else as 0.
- Backtracking tries to find a vector X that maximizes or minimizes criterion function $P(x_1, x_2, \ldots, x_n)$.

• Algorithm BACKTRACK(K)
// X[1...K-1] is the solution vector
// T(X[1], X[2], ..., X[K-1]) is the state space tree,
// Bk() is the bounding function
for each x[K] ∈ T(x[1], x[2], ..., x[K-1]) do
    if (Bk(x[1], ... x[K-1]) == TRUE) then
        if (x[1], ..., x[K-1]) is path to ans node then
            print (x[1], ... x[K-1])
        end
        if K < n then
            BACKTRACK(K+1)
        end
    end
end

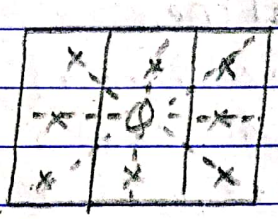
• Time Analysis:-
Performance depends on four parameters:-
1. Time to compute tuple x[K]
2. Number of x[K] which satifies the explicit constrai
3. Time taken by bounding function to generate feasible sequence.
4. Number of x[K] which satisfy the bounding function Bk to generate feasible sequence.

Complexity is :- $O(P(n)\ n!)$

- $P(n)$: Time to execute first 3 steps can be computed in polynomial time.
- $n!$: Time to execute 4th step.

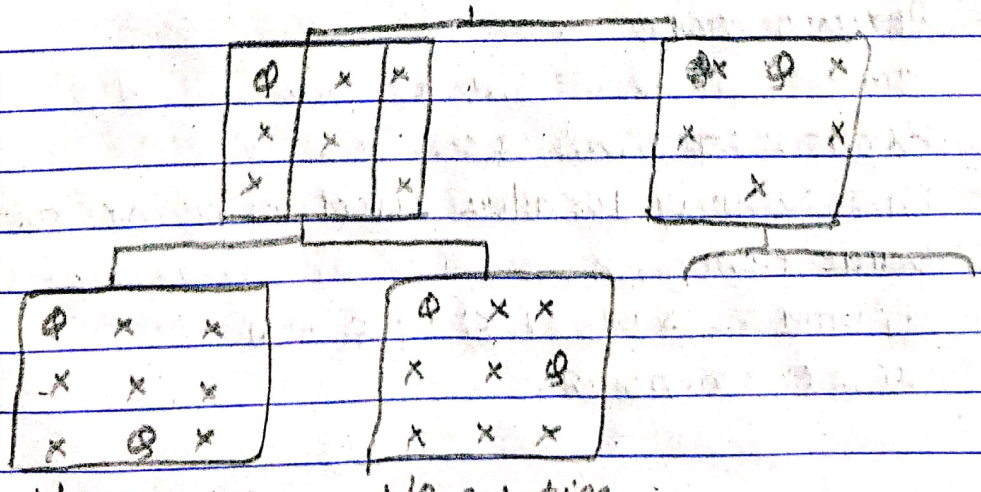
* 8-Queen Problem :-

- Problem :- Given 8x8 chess board, arrange 8 queens in such a way that no two queens attack each other.
- Two queens attack each other if they are in same row, column or diagonal.



Queen Attacks

- Eight queen problem has 92 solution out of which 12 solutions are fundamental solution meaning other 80 solutions can be constructed by rotating or reflecting these 12 solutions.
- Eight queen is a special case of N-Queen problem
- For simplicity to understand backtracking used in N-Queens consider N = 3.



No solution

No solution

- We continue placing queens either row by row or col by col and check if bounding function is meet.
- If met then we continue exploring otherwise we cut-off.

• Algorithm NQUEEN(n, k)
// Description : Solve n×n queen using backtracking
// Input : n : Number of queens
k : Number of queen processed currently

```
for i ← 1 to n do
    if PLACE(k, i) then
        x[k] = i
        if k == n then
            print x[1..n]
        else
            NQUEEN(n, k+1)
        end
    end
end
Function PLACE(k, i)
for j ← 1 to k-1 do
    if x[j] == i or ((abs(x[j] - i) == abs(j-k)) then
        return false
    end
end
return true
```

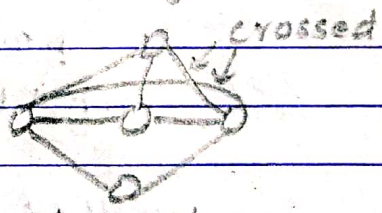
• Explicit constraints :-
- These allow/disallow selection of x_i to take value from given set. i.e $x_i = 0$ or $x_i = 1$
$x_i = 1$ if $LB_i \leq x_i \leq UB_i$
$x_i = 0$ otherwise

- Solution space is formed by collection of tuple that satisfies the constraint.
- Implicit Constraint:-
- Which of the tuple solution space satisfies the given criterion functions.
- For N-Queen it is that no two queens must be in same row, col, diagonal.
- Complexity analysis:-
- At each level branching factor decreases by 1 and creates new problem of size (n-1). With n choices it creates n problems of size (n-1).
- PLACE function determines position of queen in O(n), its called n times
- $T(n) = n * T(n-1) + n^2$, its solution is $O(n!)$
- Hence TC of N-Queens is $O(n!)$

* Graph colouring problem.

- A graph is called ~~planar~~ planar if it can be drawn on 2D plane with no two edges crossing each other

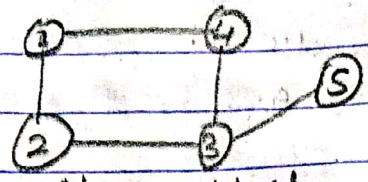
Planar



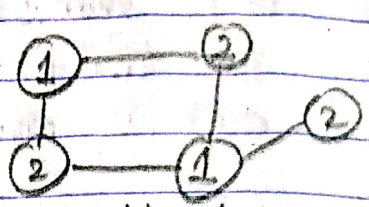
Non-planar

- Graph colouring problem is used only for planar graphs.
- Problem :- Colouring of vertices of graph such that no two graphs have same colour.
- If colouring is done using at most k-colours it is called as k-coloring.
- Smallest number of colours required for colouring graph is called its chromatic number, denoted as $\chi(G)$ it is an NP-complete problem.

- Graph colouring is a decision as well as optimization problem.

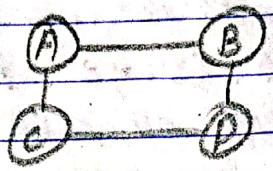


Non-optimal

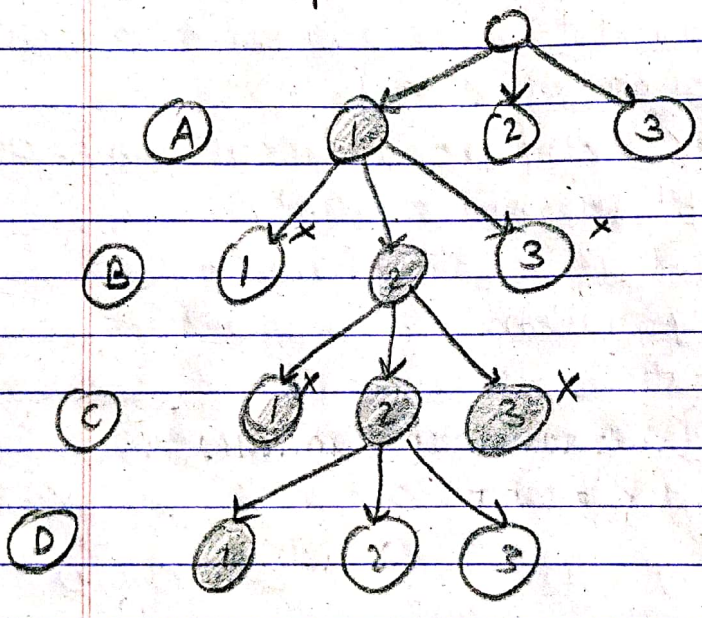


Optimal colouring

- Consider graph:



- State space tree will look like



Shaded node say colour that must be choosen.

- Algorithm..

```
Algorithm GRAPH_COLOURING(G, COLOR, i)
// G is graph of size n
// COLOUR[1...n] is array of n different colors
// i is the first vertex of graph G
If CHECK_VERTEX(i) == 1 then
    if i == N then
        print COLOR[1...n]
```

```
    else
      j ← 1
      while (j ≤ M) do
        COLOR[i+1] ← j
        j ← j+1
      end
    end
end
Algorithm CHECK_VERTEX(i)
for j ← 1 to i-1 do
  if Adjacent(i,j) then
    if COLOR[i] == COLOR[j] then
      return 0
    end
  end
end
return 1
```

- Complexity Analysis :-
  - No. of nodes increase exponentially at every level in state space tree.
  - With M colors and n vertice we have

$$T(N) = 1 + M + M^2 + \ldots + M^n = \frac{M^{n+1} - 1}{M-1}$$

$$T(N) = O(M^n)$$

* Sum of Subset Problem :-

- Problem: Given a set of positive integers find the combination of numbers that sum to give a value N.
- Numbers are sorted in ascending order.
- Solution is represented using a vector X.
- If i^{th} item can be included then x[i] = 1 else x[i] = 0

- If inclusion of one item does not violate constraints add it, otherwise backtrack remove previously added item and continue.

- Algorithm SUBSET(i, sum, N, remSum)
// i is item index
// sum is sum up till now
// N is the sum that must be achieved
// remSum = N - sum
if ~~Feasible~~ FEASIBLE(i) == 1 then
    if (sum == N) then
        print X[1 ... i]
    end
else
    X[i + 1] ← 1
    SUBSET (i+1, sum + si+1, N, remSum - si+1)
    X[i + 1] ← 0
    SUBSET (i+1, sum, N, remSum - si+1)
end
FEASIBLE(i)
if (sum + remSum ≥ N) and (sum == N) or
    (sum + si+1 ≤ N) then
    return 0
end
return 1

- Time Complexity :-
At level i we have 2^i nodes in state space tree.
$T(n) = 1 + 2 + 2^2 + \dots + 2^n = 2^{n+1} - 1 = O(2^n)$

UNIT 5

Chap 6: Branch & Bound

* Principle:-
- Backtracking used for decision problems.
- Branch n Bound used for optimization problems
- Here we build state space tree and find optimal solution by pruning few branches of tree which do not satisfy the bound.
- The branch and bound technique in which E-nodes are put in a queue is called FIFO branch & bound whereas if put in stack it is called LIFO branch & bound.
- Here bounding fⁿs are heuristic functions
- Heuristic maximizes probability of better search or minimizes probability of worst search.

• Time Analysis:-
- BnB is mostly used for combinatorial problems.
- It has exponentially increasing time complexity.

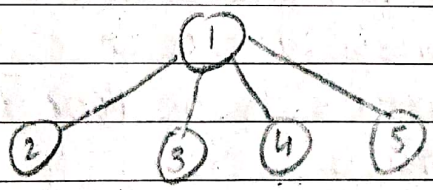
• Control Abstraction

```
Algorithm BRANCH_AND_BOUND()
E ← createNode
while (true) do
    if E is leaf node then
        print "Display path from root to E"
        return
    end
```

```
EXPAND(E)
if size(HEAP) == 0 then
    print "Solution not found"
    return
 end
 E ← deque(H)
end
Algorithm EXPAND(E)
Generate all possible child nodes of E
Compute cost of each child
Insert the childs in Heap
```

The node to be explored can be choosen in many ways :-

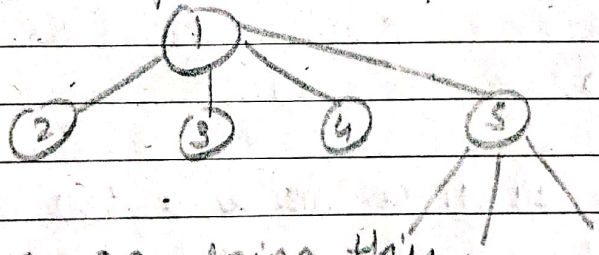
17 FIFO :-

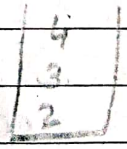


State-space Tree

Stack

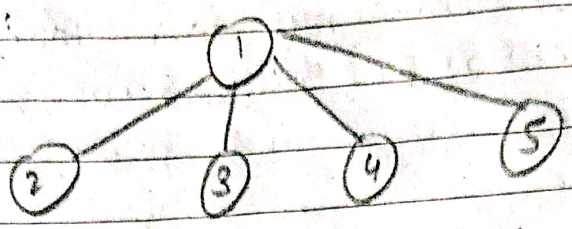
Since 5 is the top-most node we will explore it first





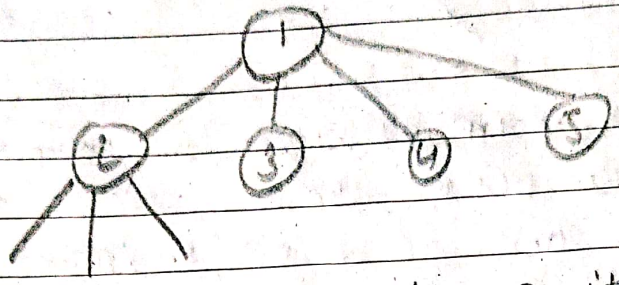
We go on doing this.
We will push expanded nodes of 5 onto stack.
The node that is being expanded is called e-node or expansion node
In FIFO method element at the top of stack will be called e-node.

2) LIFO:



[2 3 4 5]
Queue

In LIFO we use queue

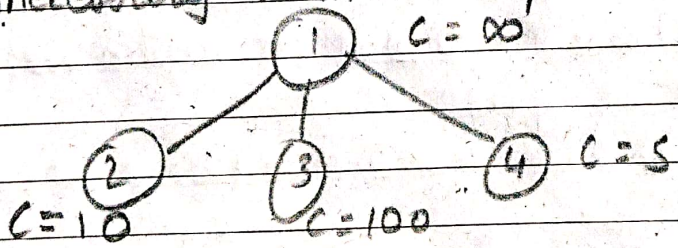


[3 4 5]
Queue

Now after expanding 2 its child nodes will go at the back of queue and 3 will be expanded.

3) least-cost Branch & Bound. (LC)

- In BnB we always use a cost function to calculate heuristics of finding an optimal and prune out unoptimal or unnecessary branch or paths.



- Out of all given nodes after expanding 1 node 4 has least cost hence we will expand it.
- Hence in LC method we expand the node that has least cost without stack/queue.

* Knapsack using Branch & Bound.

- Branch & Bound can only work minimization problems whereas 0/1 Knapsack is a profit maximization problem.
- To counter this we will choose negative upper bound and costs
- Upper bound represents the minimum cost that a live node can have, if a live node exceeds upper bound then we kill that node i.e we don't explore it anymore.
- How Upper bound:- $U = -\sum_{i=1}^{n} p_i x_i$ Cost function:- $C = -\sum_{i=1}^{n} p_i x_i$ (with Fraction)
- Consider an example of knapsack problem instance to demonstrate working of BnB.
- We are using LC BnB
- Consider,

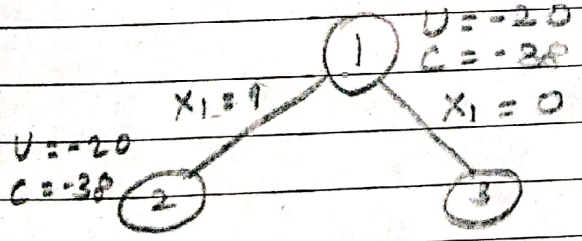
profit	10	10	12
weight	2	4	6

M = 20 [Capacity]
n = 3

1) Let us calculate costs

$C = -\left[10 + 10 + 12 \times \frac{6}{6}\right] = -38$

$U = -[+10 + 10] = -20$



2) Let us see when $x_1 = 0$

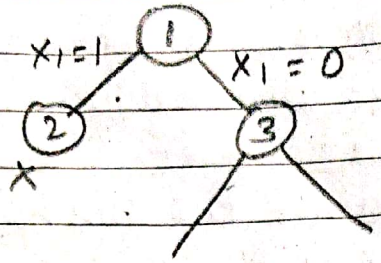
$C = -(10 + 12) = -22$

As

$U = -(10 + 12) = -22$

As $-22 < -20$ Update upper bound

Now node has upper bound -38 and new upper bound is -22 hence we will node 2

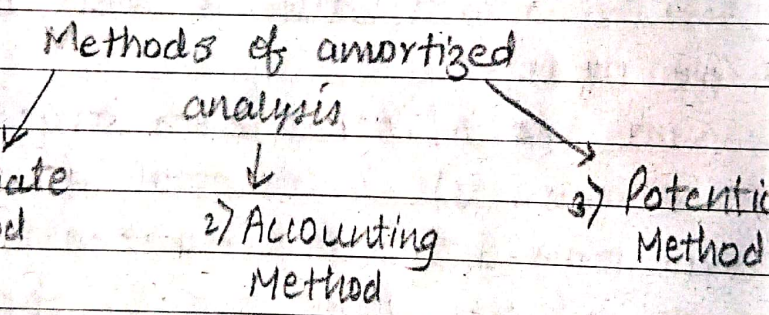


- No further objects can be included, hence we have vector solution as (0, 1, 1)
- We choose 2nd and 3rd item and profit is 22.

* Traveling Salesperson Branch-n-Bound

Chapter 7
Amortized Analysis

* Amortized Analysis:-
- Used to measure performance of algo in the avg. case.
- It measures time/resource complexity in context of comuter program.
- Some operations/resources are cheaper than others (e.g multipy expensive than addition) Amortized analysis takes this into account while measuring performance.
- It does not say anything about individual operation but rather averages the cost of all operations in the worst case.



1) Aggregate Method:-
- Assign same cost to all ~~algorithm~~ operations in the algorithm.
- We find upper bound cost $T(n)$ of n operations and calculate amortized cost as average of that $T(n)/n$.

2) Accounting Method:-
- It assigns different cost of operation for different operations which is called amortized cost which may be less than or greater than actual cost.

- If amortized cost is more than actual cost then the difference of these two is called as ~~cost~~ credit.
- Consider $\hat{c}_i$ and c_i are the amortized and actual cost of an i^{th} operation.
- Then $credit_i = \hat{c}_i - c_i$
- Total credit = $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$

- This credit is used to pay ~~dept~~ debt for operations that have more actual cost but were given less cost (i.e amortized cost).

3) Potential Method :-
- Similar to accounting but rather than credit, we use potential or energy to pay cost of operation.
- A special data structure is used to store potential of an object.
- Suppose D_0 is intial data structure and D_i is the result of i^{th} operation on data structure D_{i-1}
- A function ϕ (potential function) maps each D_i to a real value $\phi(D_i)$ which is called potential of data structure D_i.
- Amortized cost = Actual cost + Potential Difference

$$\hat{c}_i = c_i + [\phi(D_i) - \phi(D_{i-1})]$$

Amortized cost of n operation would be

$$\sum_{i=0}^{n} \hat{c}_i = \sum_{i=0}^{n} c_i + \sum_{i=0}^{n} (\phi(D_i) - \phi(D_{i-1}))$$

$$\sum_{i=0}^{n} \hat{c}_i = \sum_{i=0}^{n} c_i + [\phi(D_n) - \phi(D_0)]$$

- ϕ should be selected such that

$$\phi(D_i) \geq \phi(D_0)$$

- If potential difference > 0 then it is called overcharge and if < 0 then undercharge

* Amortized analysis of Stack:-
- The augmented stack:-
Consider stack with following operations:-
1) Push(S, x): Push x to stack S
2) Pop(S): Pop top element of S and return it
3) Multipop(S, k): Pop top k elements of stack S.
Time Complexities:- Push :- O(1)
Pop :- O(1)
Multipop :- $\theta(\min(k, s))$ = O(or O(n))

1) Aggregate Analysis of Stack:-
- The total number of pops/multipops ≤ the total number of push.
- Maximum number of push operation is n. So time for entire sequence is O(n)
- Therefore amortized cost = $\frac{O(n)}{n} = O(1)$

2) Accounting Analysis of stack:-
Consider amortized costs for methods as:-
1) Push :- 2 2) Pop :- 0 3) Multipop :- 0
- The push operation has cost 1 and an additional 1 as credit.
- There is always enough credit to pay for each operation.
- Each amortized cost is O(1)
- cost of entire sequence is O(n)

3) Potential Method:-

- Consider stack S with m elements.
- Let the potential function be $\phi(S) = m$

Now,

Push :- $\hat{c}_k = c_k + \phi(Push(S,x)) - \phi(S)$

$\hat{c}_k = 1 + 1 = 2$

Pop :- $\hat{c}_k = c_k + \phi(Pop(S)) - \phi(S)$

$\hat{c}_k = 1 - 1 = 0$

Multipop :- $\hat{c}_k = c_k - \phi(Multipop(S,k)) - \phi(S)$

$\hat{c}_k = a - a = 0$

* Amortized analysis for binary counter -
- The only operation is increment (A) which adds 1 to current number in the counter.

1) Aggregate Method :-
Total bit flips in n increment operations is
$n + n/2 + n/4 + \ldots + n/2^k < n\sqrt{1/(1-1/2)}$
$= 2n$
Total cost of sequence is $O(n)$
Amortized cost per op $= \frac{O(n)}{n} = O(1)$

2) Accounting Method :-
All changes from 1 to 0 are paid with previously stored credit.
Amortized cost per op $= O(1)$

3) Potential Method
B_i = Number of 1's after i^{th} increment.
Suppose i^{th} operation resets t_i bits
Actual cost : $c_i = t_i + 1$
Now, $B_i \leq B_{i-1} - t_i + 1$
Difference in potentials : $B_i - B_{i-1} \leq (B_{i-1} - t_i + 1) - B_{i-1}$
$= t_i - 1$
Amortized cost : $c_i + B_i - B_{i-1} \leq t_i + 1 - t_i + 1 = 2$

Chap 8:- Advanced Algorithms

* Tractable & Non-tractable:-
- If running time of a problem is polynomial then it is tractable otherwise non-tractable.

* Randomized Algos:-
- NP may be solved by randomized or approximation algorithm in polynomial time. However their solution may not be optimal.
- Goal of random algos is to quickly a soln nearer to the optimal solution.
- Categories of Randomized Algos:-
  1) Las Vegas algorithms. 2) Monte Carlo Algorithms

1) Las Vegas: For same input always produces same output. Always finds correct solution.
2) Monte Carlo:- For same input we get different output. May produce wrong answer. It is said to p-correct Monte Carlo if it produces an answer with a probability of being correct as p.

- Randomized Quicksort:-

```
Algorithm RARTITION (A, p, r)
    q ← RANDOM(p, r)
    swap(A[q], A[r])
    return partition (A, p, r)
end
Algorithm QSORT(A, p, r)
    if p<r then
        q ← PARTITION(A, p, r)
        QSORT(A, p, q-1)
        QSORT(A, q+1, r)
end
```

* Approximate Algorithms:-
- Quick solution but not guarnteed optimal soln.
- Apply heuristics to cut computations.
- Feasible soln near to optimal soln is called as approximate solution.

* Characterizing Approximation Algorithms:-
- let I be an instance of problem P.
- Now FO(I) and FA(I) represent value of optimal and feasible solution generated by algo A.
- Based on their difference we categorize approximation algorithm, which is as follows:-

1) $|FO(I) - FA(I)| \leq k$ [Absolute Approximation]
2) $(|FO(I) - FA(I)| / FO(I)) \leq f(n)$ [$f(n)$ - approx^n]
3) $|FO(I) - FA(I)| / FO(I) \leq \epsilon$ [ϵ approx^n]

- Performance Ratio:-
Let C^* be cost of optimal soln and C be cost of soln returned by approx^n algorithm as C, Approximation ratio P(n) for input size n is given as,

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

* Embedded Algorithms:-
- Embedded Systems:-
- Designed for specific functions only.
- Tightly coupled between software and hardware.
- These are time-bound.

- Embedded System Scheduling (Power Optimized Scheduling Algorithm)
- Scheduling Algos can be classified as:-

1) Fixed priority algorithms:-
- Priority of task assigned at design time.
- Task is moved to exit, blocked or waiting state.
- Simple lifecycle of task is followed here.

2) Dynamic priority algorithm:-
- Priority of task changes dynamically.
- Rest is same as fixed priority.

• Sorting algorithm for embedded systems.
- Sorting algos for Embedded systems should have the following characteristics:-
1) It should be in-place.
2) Algo should be non-recursive.
3) It should work in resonably less time.
- Insertion sort is used in embedded systems

UNIT 6
Multithreaded & Distributed Algorithms

• Dynamic Multi-threading:-
- Allows programmer to specify parallelism in the application without worrying about communication protocols, load balancing, etc.
- Scheduler takes care of load balancing.
- All dynamic threading supports two features:-
1) Nested parallelism:- Allows a subroutine to spawn and allows caller to proceed parallel.
2) Parallel loops:- Simple loops executed parallelly.

Modes of Communication in parallel Computing

↙ Shared Memory:
: All processors access common memory.

↘ Distributed Memory:
: All processors have their local mem.

Page No.
Date

- Serial algos can be made parallel by adding keywords parallel, spawn and sync.
- Multi-threaded computation can be represented using DAG, it is called as computation graph.

* Performance Measures:-
- A sequence of instructions that have no parallel control can be grouped into a single strand.
- The strand of maximum length will be called as thread.
- Total time taken to execute on a single processor is work. Work is summation of time taken by each strand.
- Longest time to execute a strand along any path in a computation dag is called as span.

- Consider T_p as running time of an algo on processor P, let T_i represent time on i^{th} processor.
- If we have unlimited processors then the span will be T_∞.
- P processors perform P amount of work in unit time, in time T_p, they will perform $T_p \times P$ work.
- Let T_1 be the total work to be done:-

$\therefore$ ~~$T_p \cdot P \leq T_1$~~ $T_p \cdot P \geq T_1$

$\therefore T_p \geq \frac{T_1}{P}$ ... (work Law)

- A machine with P processors cannot run faster than machine with unlimited processors. However a machine with unlimited processors can emulate a machine with P processors. Hence

$T_p \geq T_\infty$ ... (Span Law)

- Speed up of Computation : $\frac{T_1}{T_p}$ | Parallelism : $\frac{T_1}{T_\infty}$

Page No.
Date

Slackness = $\frac{T_1}{(PT_\infty)}$

∴ Factor by which the parallelism of computation exceeds the number of processors in the machine.

* Analyzing Multithreading Algorithms :-

- Two points should be considered while analyzing multi-threading algorithms:-
  1) Analysis of work, 2) Analysis of spawn
- To demonstrate consider a parallel fibonacci computation:

```
Fibonacci(n):
    if n<2 then return n;
    x = spawn Fibonacci(n-1);
    y = spawn Fibonacci(n-2);
    sync;
    return x+y;
```

- Work of Fibonacci:-
- Work done here is same as work done in serial version of fibonacci which is ϕ^n where $\phi^n = \left(\frac{1+\sqrt{5}}{2}\right)^n$

∴ $T(n) = T_1(n) = \theta(\phi^n)$.

- Span of Fibonacci:-

$T_\infty = \max(T_\infty(n-1), T_\infty(n-2)) + \theta(1)$

$T_\infty = T_\infty(n-1) + \theta(1)$

∴ $T_\infty = \theta(n)$

- Parallelism of fibonacci:-

$\frac{T_1(n)}{T_\infty(n)} = \theta\left(\frac{\phi^n}{n}\right)$

Page No.
Date

- Parallel Loops :-
We simply use parallel keyword before for keyword to make it run parallelly

* Race conditions :-
- Multithreaded algo is deterministic if and only if does the same thing on input no matter how the inputs are scheduled.
- It is non-deterministic if the case is opposite.

- Determinacy Race :-
- Occurs when two logically parallel instructions access the same memory and atleast one of them is a write instruction.

* Multi-threaded Matrix Multiplication :-

```
Algorithm PMM(A, B)
parallel for i ← 1 to n do
    parallel for j ← 1 to n do
        cij ← 0
        for k ← 1 to n do
            cij ← cij + aik * bkj
        end
    end
end
return C
```

$T_\infty = \Theta(\log n) + \Theta(\log n) + \Theta(n) = \Theta(n)$

$\frac{T_1}{T_\infty} = \frac{\Theta(n^3)}{\Theta(n)} = \Theta(n^2)$

* Multi-threaded merge sort:-

```
Algorithm PMS (A, low, high)
  If low < high then
    mid ← ⌊(low+high)/2⌋
    spawn PMS(A, low, mid)
    spawn PMS(A, mid+1, high)
    sync
    MERGE(A, low, mid, high)
end
```

$T_1 = \Theta(n \log n)$ $T_\infty = T_\infty(n/2) + \Theta(n) = \Theta(n)$

$$\frac{T_1}{T_\infty} = \frac{\Theta(n \log n)}{\Theta(n)} = \Theta(\log n)$$

* Distributed algorithm:-
- Algorithms work on multi-processors and they do not share memory as is in the case of multi-threading.

- Distributed BFS
- Distributed MST.

Ch. 10: String Matching Algorithm

* Naive String Matching:-

```
Algorithm match (T, P)
  for i ← 0 to n-m do
      if P[1...m] == T[i+1, ... i+m] then
          print "Pattern Found"
      end
end
```

TC:- $O(m * (n-m))$

m :- length of pattern

n :- Total length of string

```
x ← Hash (P)
for i←0 to n-m do:
    y ← Hash (T[i+1...i+m]
    if x==y:
```

Page No.
Date

* Rabin-Karp algorithm:-
- It also uses sliding window to match pattern like naive algo.
- However only if hash value of pattern matches with the hash value of current substring we start matching the individual characters.
- So Rabin-Karp needs two things:-
  i) Hash value of pattern
  ii) Hash value of all substrings of length m.
- Algorithm RK(T, P)

```
    x ← HASH(P[1...m])
    for i←0 to n-m do
        y ← HASH(T[i+1...i+m])
        if x == y then
            if P[1..m] == T[i+1...i+m] then
                print "Pattern Found"
            end
        end
    end
print "Pattern not found"
```