Greedy Algorithms (G.A)

G.A used for solving Optimization problems (in which you find not just a Solution but the best one).

Goal :- To find best solution among a set of possible solutions.

★ G.A are powerful to solve optimization problems But they are not alway applicable or guaranteed to provid optimal solution. But when they do, they're Simplest & efficient algos available.

Example :-

Activity Selection Problem.

Activity Scheduling is a simple scheduling problem for which greedy approach provides an optimal solution.

Problem Statement :-

- We are given a set $S = \{a_1, a_2, \ldots a_n\}$ of n activities that are to be scheduled to use some resource.
- Each activity a_i must be started at a given start time s_i and ends at finish time f_i.
- There is only one resource.
- Some start and finish time may overlap. Therefore not all requests can be honored.
- Two activities a_i and a_j are non-interfering if they do not overlap

$$(s_i, f_i) \cap (s_j, f_j) = \phi$$

Goal :-

We've to select a maximum size set of mutually non-interfering activities to use that resource.

How do we schedule max. no of activities to use a resource?

1 - A Solution you can think of (But its not optimal one)

Intuitively, we don't like long activities Because they occupy resource for long time and keep us away from honoring other requests.

Greedy Choice :-

Repeatedly → Select activities with smallest duration ($f_i - s_i$) and schedule it provided that it does not overlap with any previously scheduled activities.

⟷

Not optimal

Here is a **Counter Example**

Activities	Start_Time	Finish-Time
A	4	6
B	1	5
C	5	7

Activity-A has duration of '2' (Say hours)

Activity-B ---------- 4 hours

Activity C ---------- 2 hours.

* Sort activities according to duration

Now If we select Activity-A first and schedule it (having smallest duration) Then other activities will not be Scheduled because they both overlap with it

So

Scheduled Activities Set $= \{A\}$

But optimal Solution is

$\{B, C\}$

Counter Example

So

we didn't make greedy choice correctly.

Another Solution :- (Greedy Algorithm)

- Sort the activities by their finish time
- Greedy choice
- Select the activity that finishes first and schedule it.
- Now Select activities one by one that do not interfere with already scheduled activities.

```
ActivityScheduling ( A , n)
{
    Sort 'A' by finish times
    S = { A[1] }        // schedule first activity

    Prev ← 1
    for ( i ← 2 TO n)
    {  if ( A[i].start >= A[Prev].finish)
       {  S ← S U { A[i] }
          Prev ← i
       }
    }
```

Time is dominated by sorting

Thus, Complexity is $O(n \log n)$

Proof of Correctness

Actually Greedy Algorithms Provide optimal solution for those problems that exhibits:

1- Greedy choice property:
(selecting Local optimum at each st

2- Optimal substructure:
(Global optimal solution can be constructed from optimal solutions of its Subproblems)

So

- Our Proof of Correctness is based on showing that the first choice mad by algorithm is best possible.
- And then using induction to show that the algorithm is globally optimal.

★ We'll always make some solution (optimal) then transform into greedy solution without increasing cost.

Prod For Greedy Choice Property:

Claim:
Let $S = \{a_1, a_2, \ldots, a_n\}$ 'n' activities sorted by increasing finish times need to be scheduled to use some resource

Then there is an optimal solution in which a_1 is scheduled first.
greedy choice

Proof :-

Let "A" be an optimal schedule and Let "x" be the activity in A with smallest finish time.

- if $(x == a_1)$ then we are done.
- Otherwise we form a new schedule A' by replacing x with a_1

(new Schedule should be feasible)

$A' = A - \{x\} \cup \{a_1\}$ is feasible

Because
$A - \{x\}$ can not have any activities start before "x" finishes otherwise they will interfere with "x"

And
Since a_1 is by definition first to finish. so it has an earlier finish time than "x"

Thus, a_1 can not interfere with any activities in $A - \{x\}$

So A' is feasible schedule.

Clearly,
A and A' contain the same number of activities.
Implying that A' is also optimal.

Both A and A' are optimal But. A' uses greedy Approach and select a_1 as first activity and ~~new~~ schedule A can be converted into A' (just be replacing x with a_1)

Hence greed choice made by algorithm is optimal

So you're left with smaller and smaller subpro2.

Proof for Optimal Substructure

Claim:
Greedy algorithm gives an optimal Solution to activity Scheduling problem.

Proof :-

Base Case:-
if there are no activities or there is just one activity then greedy Algo is trivially optimal.

Inductive Case:-

Inductive Hypothesis :- Let us assume that greedy algo is optimal on any set of activities of size smaller than |S|.

And
we have to prove it optimal for "S"

Let S' be the set of activities that do not interfere with a_1.

$$S' = \{ a_i \in S \mid Start_i \geq finish_1 \}$$

- Any Solution for S' can be converted to Solution for "S" by Simply adding activity a_1 and vice versa.

- Activity a_1 is optimal schedule (by previous claim)

- it follows that to produce an optimal schedule for overall problem, we first schedule a_1 and append optimal schedule for S'. (for subproblem)

So optimal Solution S' transformed into global optimal Solution S just by appending (no extra cost)

Hence
Proved that greedy algorithm is providing optimal Solution to ASP.

Fractional Knapsack Problem

Problem Statement:-

- Given a set of "n" items, each with a weight "wi" and a value "vi",
- Knapsack capacity is "W"

Goal:-

Determine the most valuable combination of items to include in a knapsack with a limited capacity.

- Items can either be put in the knapsack or not
- One is allowed to take fraction of ~~value~~ an item for a fraction of weight and fraction of value.

★ If you make greedy choice that Pick item having less weight first

★ If we make greedy choice that pick the item having max value first.

Both choices will not end up with optimal solution. So

Greedy Choice:- Calculate value-to-weight ratio and pick item that has max ratio. (Repeatedly)

Algorithm

```
fractionalKnapsack(items, n, W)
{
    Sort items by value-to-weight ratio in decreasing order
    valueSum = 0, frac = 1
    for(i ← 0 To n-1)
    {
        if (W == 0)
            return valueSum;
        if (wi ≥ W)
            frac = W/wi
        valueSum += frac * Vi
        W = W - frac * wi
    }
    return valueSum;
}
```

Time Complexity dominated by Sorting so it is $O(n \log n)$

Proof of Correctness

In order to show the correctness of a greedy algorithm, we need to prove that the following two properties hold:

1 - Greedy choice property.
2 - Optimal Substructure.

Prove Greedy Choice Property:

Let there is an optimal Solution $A = \{a_1, a_2, \dots a_n\}$ to fractional Knapsack problem.
And it does not include item "i" with max ratio. (v_i/w_i)
a_i indicates fraction of i^{th} item.
Suppose a_1 in A has highest ratio

$$\frac{v_{a_i}}{w_{a_i}} \geq \frac{v_{a_1}}{w_{a_1}}$$

Now if we remove a_1 then Solution is

$$A' = A - \{a_1\}$$

$$= w_{a_i} \times \frac{v_{a_1}}{w_{a_1}} + w_{a_1} \times \frac{v_{a_i}}{w_{a_i}} = w_{a_1}\left(\frac{v_{a_i}}{w_{a_i}} - \frac{v_{a_1}}{w_{a_1}}\right) > 0$$

Now combine A' with i^{th} item (or fraction of it) we'll have greater or equal valuable solution B

$$B = A' \cup \{a_K\}$$

if B is better than A:

this is contradiction and i^{th} item must be included.

if B = A:

Greedy choice already included anyway.

(Since we are taking out weight "w_{a_1}" and adding Same weight of i^{th} item (as whole or fraction of it)

So we are still within capacity.

So our optimal solution is not so optimal. we have to transform it to greedy one by including item having

Date :

Optimal Substructure Proof (Fractional knapsack)

Assume that **A** is optimal Solution with value **V** to problem **S** with knapsack capacity **W**.

Then we want to prove that $A' = A - \{a_1\}$ is an optimal Solution with V' to the subproblem $S' = S - \{a_1\}$ and knapsack capacity $W' = W - w_1$.

(where a_1 item has highest value-to-weight ratio)

Proof by Contradiction:

we assume A' is not optimal to S' and we have another Solution A'' to S' that has higher total value, $V'' > V'$.

So $A'' \cup \{a_1\}$ is a Solution to problem S with value $V'' + V_1 > V' + V_1 = V$

So $V'' + V_1 > V$ contradicts our assumption that V is optimal in the beginning.

Two conditions for a problem to be solved using DP.

1- overlapping Subproblems.
2- Optimal Substructure.

Date :

Dynamic Programming

In DP, problems are solved by breaking them down into simpler subproblems and solving each subproblem only once, storing the solution to avoid redundant calculations.

☆ It is essentially recursion without repetition

Steps :-

To develop a dynamic prog. algorithm, two key steps are involved.

1- Formulate problem recursively.
(write formula for whole prob. as simple combination of smaller subproblems)

2- Build Solution to above recurrence (using tabulation)
(write algo that starts with base case and works its way up to the final solution)

☆ DP algos store results of subproblems in a table (but not alway

BERGER PAINTS

Date :

Motivation for DP (Fibonacci Seq)

- Put a pair of rabbits in a room
- How many pairs of rabbits can be produced from that in a year.

if

* Every month each pair begets a new pair
* Each new pair will mature for one month and then produce a new pair in second month.

Resulting Sequence :-

1, 1, 2, 3, 5, 8, 13, ...

Each number is sum of the two preceding numbers.

This Sequence is called Fibonacci Sequence.

Because this problem was posed by Leonardo Pisano known as Fibonacci in his book (Liber Abaci) published in 1202.

Example #01

BERGER PAINTS

Date :

Recursive Algo To find n^{th} fibonacci term.

```
fib(n)
{
  if (n < 2)
      return n;
  return fib(n-1) + fib(n-2)
}
```

In this algo a lot of calls are repetetive. so Time complexity is

$$T(n) = T(n-1) + T(n-2) + 1$$

$$T(n) = O(2^n)$$

which is exponential.

We can avoid repetitive calls by storing results of recursive call and Looking them up again, if we need them Later. (Memorizati

Recurrence: $fib(i) = \begin{cases} i & i<2 \\ fib(i-1)+fib(i-2) & \end{cases}$

Memoization

```
memoFib(n)
{
    if (n<2)
        return n
    if (F[n] == -1)
        F[n] = memoFib(n-1) + memoFib(n-2)
    return F[n]
}
```

F | -1 | -1 | -1 | ...

Tabulation

```
tabuFib(n)
{
    F[0] = 0
    F[1] = 1
    for(i ← 2 To n)
        F[i] = F[i-1] + F[i-2]
    return F[n]
}
```

$T(n) = O(n)$

Example #02

Maximum Subarray Sum (Kadane's Algo)

```
maxSubarraySum(A, n)
{
    curSum = overAllSum = A[0]
    for(i ← 1 To n-1)
    {
        if (curSum >=0)
            curSum += A[i]
        else
            curSum = A[i]
        if (curSum > overAllSum)
            overAllSum = curSum
    }
    return overAllSum
}
```

Time Complexity = $O(n)$

Recurrence: Base case ⇒ $M_0 = A[0]$

$M_i = \max\{A[i], M_{i-1} + A[i]\}$

Example #03

BERGER PAINTS

Date :

Edit Distance

* Algo was introduced in 1966 by Molecular biologists.

⇒ Edit distance problem is used to measure the similarity b/w two strings.

- It calculates minimum number of operations required to transform one string to another.

Allowed Operations

Maintain (M), Insertion (I), deletion (D), Substitution (S),

Example:-

if we ~~write~~ type a word which is not present in dictionary then google / ms word / cell phone give us some suggestions close to that word

(Suggestions are taken by calculating edit distaces. And words having less edit distance to typed word are Suggested

Step-1 (Formulate Recurrence)

- Suppose we have an **m**-character string A and an **n** character string B
- Define $E(i,j)$ to be the edit distance b/w first i characters of A and j characters of B
- The edit distance b/w entire strings 'A' and 'B' is $E(m,n)$

Optimal Substructure:

If we remove last column, the remaining columns must represent the shortest edit sequence for remaining substrings.

Let's denote two input strings $A[1..i]$ and $B[1..j]$, where i,j represents their lengths, Then

Recurrence formula:

$$E(i,j) = \begin{cases} 0 & ; \text{ if } i=0 \text{ \& } j=0 \quad (\text{Both empty}) \\ i & ; \text{ if } j=0 \quad (\text{B string is empty}) \\ j & ; \text{ if } i=0 \quad (\text{A is empty.}) \end{cases}$$

$A[i] = A[j]$ if there is match

$$E(i,j) = \begin{cases} \min \begin{cases} E(i-1,j-1) & \text{if there is match} \\ E(i-1,j-1)+1 & \text{Substitute} \\ E(i-1,j)+1 & \text{deletion} \\ E(i,j-1)+1 & \text{insertion} \end{cases} \end{cases}$$

In our example, we have to find edit distance of 'ARTS' and "MATHS"

So we need to compute $E(5,4)$

then

$$E(5,4) = \begin{cases} E(4,3) \\ E(4,3)+1 \\ E(4,4)+1 \\ E(5,3)+1 \end{cases}$$

Recursion leads to same repetitive calls.

To avoid this we'll use DP (Tabulation)

DP Solution

```
editDistance (m, n, A, B)
{
    [redacted] ed[m+1][n+1]
    for (i ← 0 To m)
        ed[i][0] = i
    for (j ← 0 To n)
        ed[0][j] = j
    for (i ← 1 To m)
    {   for (j ← 1 To n)
        {   if (A[i-1] == B[j-1])
                ed[i][j] = ed[i-1][j-1]
            else
                ed[i][j] = 1 + min(ed[i-1][j], ed[i][j-1], ed[i-1][j-1])
        }
    }
    return ed[m][n]
}
```

Time Complexity ⇒ $O(m*n)$

if $n == m$

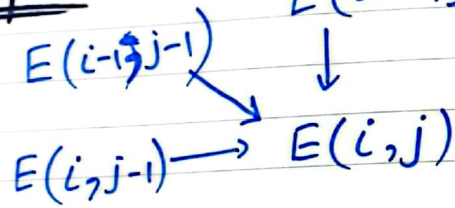
⇒ $O(n^2)$

Naive Solution

```
editDistance (A, m, B, n)
{   if (m == 0) return n
    if (n == 0) return m
    if (A[m-1] == B[n-1])
        return editDistance(A, m-1, B, n-1)
    return 1 + min(
        editDistance(A, m, B, n-1),
        editDistance(A, m-1, B, n),
        editDistance(A, m-1, B, n-1))
}
```

Time Complexity ⇒ $O(3^n)$

Step-2 :- (Tabulation)



To fill first row and first column, use base cases.

		A	R	T	S
	0→	1	→2	→3	→4
M	1	1	→2	→3	→4
A	2	1	2	→3	→4
T	3	2	2	2	→3
H	4	3	3	3	3
S	5	4	4	4	3

minimum cost

Solution Path-1

Edit transcript (ET)

D M S S M ←

M A T H S

– A R T S

+1 +0 +1 +1 +0 = 3

Solution Path-2

D M I M D M ←(ET)

M A – T H S

– A R T – S

+1 +0 +1 +0 +1 +0 = 3

Solution Path-3

S S M D M ←(ET)

M A T H S

A R T – S

+1 +1 +0 +1 +0 = 3