Algorithmic paradigm: Introduction

An **algorithmic paradigm** or **algorithm design paradigm** is a generic model or framework which underlies the design of a class of algorithms. An algorithmic paradigm is an abstraction higher than the notion of an algorithm, just as an algorithm is an abstraction higher than a computer program.

- Backtracking
- Branch and bound
- Brute-force search
- Divide and conquer
- Dynamic programming
- Greedy algorithm
- Prune and search

Parameterized complexity

- Kernelization
- Iterative compression

Computational geometry

- Sweep line algorithms
- Rotating calipers
- Randomized incremental construction.
-
    - **Divide and Conquer**
    - Idea: Divide problem instance into smaller sub-instances of the same problem, solve these recursively, and then put solutions together to a solution of the given instance.
    - Examples: Merge sort, Quick sort, Strassen's algorithm, FFT.
    - **Greedy Algorithms**
    - Idea: Find solution by always making the choice that looks optimal at the moment — don't look ahead, never go back.
    - Examples: Prim's algorithm, Kruskal's algorithm.
    - **Dynamic Programming**
    - Idea: Turn recursion upside down.
    - Example: Floyd-Warshall algorithm for the all pairs shortest path problem.

Note: The word paradigm does translate to example, but that's not how it's used in a scientific context. All examples of algorithms (except the travelling salesman problem, which is a NP-hard problem), none of which is trivial enough to be considered an algorithmic paradigm.

Common design Algorithmic Paradigms:

- **Divide and conquer :** Recursively breaking down a problem into two or more sub-problems of the same (or related) type.

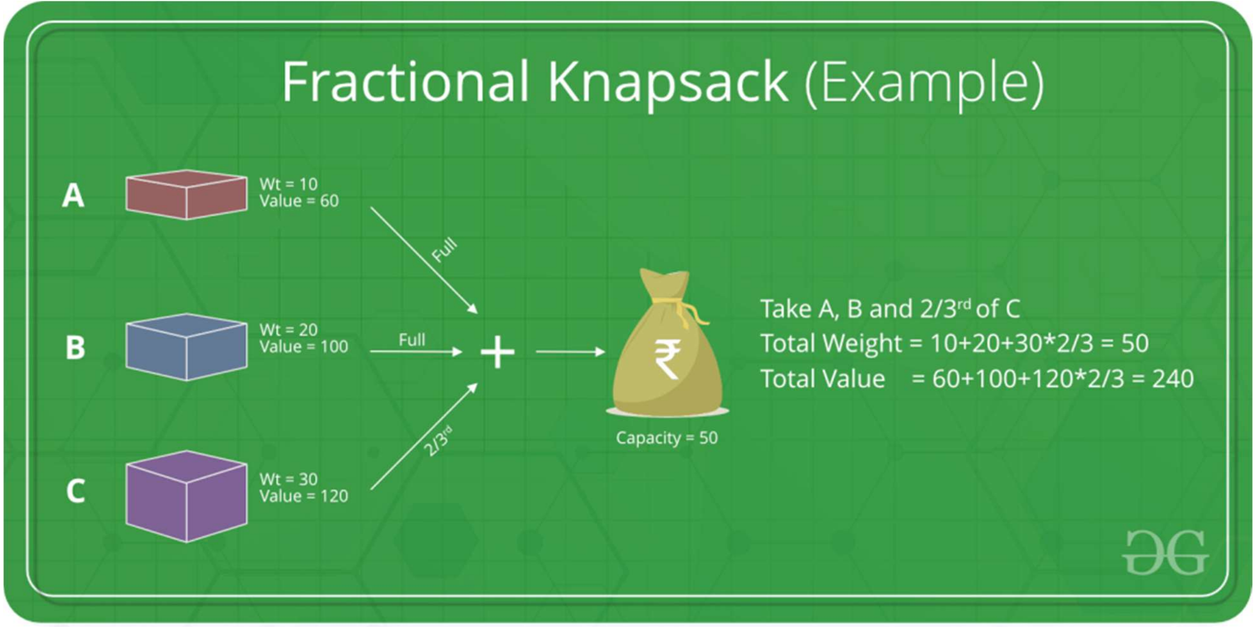
- **Dynamic programming :** breaking it down into a collection of simpler subproblems. Example: Tower of Hanoi puzzle
- **Greedy algorithm :** the problem solving heuristic of making the locally optimal choice at each stage. Example: traveling salesman problem
- **Backtracking :** is a general algorithm for finding all (or some) solutions to some computational problems Example: Sudoku puzzle solved by backtracking.
- **Brute Force :** a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

Greedy Algorithms

- Greedy is an algorithmic paradigm that **builds up a solution piece by piece**, always **choosing the next piece that offers the most obvious and immediate benefit**.

- So the problems where choosing locally optimal also leads to global solution are best fit for Greedy.
- For example consider the Fractional Knapsack Problem.
- The local optimal strategy is to choose the item that has maximum value vs weight ratio.
- This strategy also leads to global optimal solution because we allowed to take fractions of an item.

- Fractional Knapsack Problem



Fractional Knapsack (Example)

Take A, B and 2/3rd of C
Total Weight = 10+20+30*2/3 = 50
Total Value = 60+100+120*2/3 = 240

Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don't take it.

Input:
Items as (value, weight) pairs
arr[] = {{60, 10}, {100, 20}, {120, 30}}
Knapsack Capacity, W = 50;
Output:
Maximum possible value = 240
by taking items of weight 10 and 20 kg and 2/3 fraction
of 30 kg. Hence total price will be 60+100+(2/3)(120) = 240

In Fractional Knapsack, we can break items for maximizing the total value of knapsack. This problem in which we can break an item is also called the fractional knapsack problem.

```
Input :
Same as above
Output :
Maximum possible value = 240
By taking full items of 10 kg, 20 kg and
2/3rd of last item of 30 kg
```

A **brute-force solution** would be to try all possible subset with all different fraction but that will be too much time taking.

An **efficient solution** is to use Greedy approach.

- The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio.
- Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can.
- Which will always be the optimal solution to this problem.
- A simple code with our own comparison function can be written as follows, please see sort function more closely, the third argument to sort function is our comparison function which sorts the item according to value/weight ratio in non-decreasing order.
- After sorting we need to loop over these items and add them in our knapsack satisfying above-mentioned criteria.

Below is the implementation of the above idea:

```
// C/C++ program to solve fractional Knapsack Problem

#include <bits/stdc++.h>

using namespace std;

  // Structure for an item which stores weight and

// corresponding value of Item

struct Item

{

    int value, weight;

      // Constructor

    Item(int value, int weight)

        : value(value)

        , weight(weight)

    {

    }

};

  // Comparison function to sort Item according to val/weight

// ratio

bool cmp(struct Item a, struct Item b)

{
    double r1 = (double)a.value / (double)a.weight;

    double r2 = (double)b.value / (double)b.weight;

    return r1 > r2;

}

  // Main greedy function to solve problem

double fractionalKnapsack(int W, struct Item arr[], int n)

{

    //    sorting Item on basis of ratio

    sort(arr, arr + n, cmp);

      //    Uncomment to see new order of Items with their

    //    ratio

    /*

    for (int i = 0; i < n; i++)

    {

        cout << arr[i].value << "  " << arr[i].weight << " :

    "

             << ((double)arr[i].value / arr[i].weight) <<

    endl;

    }

    */
```

```
int curWeight = 0; // Current weight in knapsack

double finalvalue = 0.0; // Result (value in Knapsack)

  // Looping through all Items

for (int i = 0; i < n; i++)

{

    // If adding Item won't overflow, add it completely

    if (curWeight + arr[i].weight <= W)

    {

        curWeight += arr[i].weight;

        finalvalue += arr[i].value;

    }

      // If we can't add current Item, add fractional part

    // of it

    else

    {

        int remain = W - curWeight;

        finalvalue

            += arr[i].value

               * ((double)remain / (double)arr[i].weight);
            break;

        }

    }

      // Returning final value

    return finalvalue;

}

  // Driver code

int main()

{

    int W = 50; //    Weight of knapsack

    Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };

      int n = sizeof(arr) / sizeof(arr[0]);

      // Function call

    cout << "Maximum value we can obtain = "

         << fractionalKnapsack(W, arr, n);

    return 0;

}
```

Output

```
Maximum value we can obtain = 240
```

Greedy Algorithm to find Minimum number of Coins

Given a value V, if we want to make a change for V Rs, and we have an infinite supply of each of the denominations in Indian currency, i.e., we have an infinite supply of { 1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, what is the minimum number of coins and/or notes needed to make the change?

Examples:

```
Input: V = 70
Output: 2
We need a 50 Rs note and a 20 Rs note.

Input: V = 121
Output: 3
We need a 100 Rs note, a 20 Rs note and a 1 Rs coin.
```

Solution: Greedy Approach.

Approach: A common intuition would be to take coins with greater value first. This can reduce the total number of coins needed. Start from the largest possible denomination and keep adding denominations while the remaining value is greater than 0.

Algorithm:

1. Sort the array of coins in decreasing order.
2. Initialize result as empty.
3. Find the largest denomination that is smaller than current amount.
4. Add found denomination to result. Subtract value of found denomination from amount.
5. If amount becomes 0, then print result.
6. Else repeat steps 3 and 4 for new value of V.

```
// C++ program to find minimum

// number of denominations

#include <bits/stdc++.h>

using namespace std;

 // All denominations of Indian Currency
```

```
int deno[] = { 1, 2, 5, 10, 20,

               50, 100, 500, 1000 };

int n = sizeof(deno) / sizeof(deno[0]);

void findMin(int V)

{

    sort(deno, deno + n);

    // Initialize result

    vector<int> ans;

    // Traverse through all denomination

    for (int i = n - 1; i >= 0; i--) {

        // Find denominations

        while (V >= deno[i]) {

            V -= deno[i];

            ans.push_back(deno[i]);

        }

    }

    // Print result

    for (int i = 0; i < ans.size(); i++)

        cout << ans[i] << " ";
}

// Driver program

int main()

{

    int n = 93;

    cout << "Following is minimal"

         << " number of change for " << n

         << ": ";

    findMin(n);

    return 0;

}
```

Output:

```
Following is minimal number of change
for 93: 50  20  20  2  1
```

Complexity Analysis:

- **Time Complexity:** O(V).
- **Auxiliary Space:** O(1) as no additional space is used.

Note:

- The above approach may not work for all denominations.
- For example, it doesn't work for denominations {9, 6, 5, 1} and V = 11.
- The above approach would print 9, 1 and 1. But we can use 2 denominations 5 and 6.

  For general input, below dynamic programming approach can be used:

Find minimum number of coins that make a given value

Given a value V, if we want to make change for V cents, and we have infinite supply of each of C = { C1, C2, .. , Cm} valued coins, what is the minimum number of coins to make the change?

Examples:

```
Input: coins[] = {25, 10, 5}, V = 30

Output: Minimum 2 coins required

We can use one coin of 25 cents and one of 5 cents

Input: coins[] = {9, 6, 5, 1}, V = 11

Output: Minimum 2 coins required

We can use one coin of 6 cents and 1 coin of 5 cents
```

This problem is a variation of the problem discussed **Coin Change Problem.**
Here instead of finding total number of possible solutions, we need to find the solution with minimum number of coins.
The minimum number of coins for a value V can be computed using below recursive formula.

```
If V == 0, then 0 coins required.

If V > 0

   minCoins(coins[0..m-1], V) = min {1 + minCoins(V-coin[i])}

                                where i varies from 0 to m-1

                                and coin[i] <= V
```

Below is recursive solution based on above recursive formula.

```
// A Naive recursive C++ program to find minimum of coins

// to make a given change V

#include<bits/stdc++.h>

using namespace std;

 // m is size of coins array (number of different coins)
```

```
int minCoins(int coins[], int m, int V)

{

   // base case

   if (V == 0) return 0;

    // Initialize result

   int res = INT_MAX;

    // Try every coin that has smaller value than V

   for (int i=0; i<m; i++)

   {

     if (coins[i] <= V)

     {

         int sub_res = minCoins(coins, m, V-coins[i]);

          // Check for INT_MAX to avoid overflow and see if

         // result can minimized

         if (sub_res != INT_MAX && sub_res + 1 < res)

            res = sub_res + 1;

     }

   }

   return res;
}

 // Driver program to test above function
int main()
{
    int coins[] =  {9, 6, 5, 1};
    int m = sizeof(coins)/sizeof(coins[0]);
    int V = 11;
    cout << "Minimum coins required is "
         << minCoins(coins, m, V);
    return 0;
}
```

Output:

```
Minimum coins required is 2
```

- The time complexity of above solution is exponential.
- If we draw the complete recursion tree, we can observe that many sub problems are solved again and again. For example, when we start from V = 11, we can reach 6 by subtracting one 5 times and by subtracting 5 one times.
- So the subproblem for 6 is called twice.

Since same sub problems are called again, this problem has Overlapping Sub problems property. So the min coins problem has both properties of a dynamic programming problem.

Below is Dynamic Programming based solution.

```
// A Dynamic Programming based C++ program to find minimum of coins

// to make a given change V

#include<bits/stdc++.h>

using namespace std;

// m is size of coins array (number of different coins)

int minCoins(int coins[], int m, int V)

{

    // table[i] will be storing the minimum number of coins

    // required for i value.  So table[V] will have result

    int table[V+1];

    // Base case (If given value V is 0)

    table[0] = 0;

    // Initialize all table values as Infinite

    for (int i=1; i<=V; i++)

        table[i] = INT_MAX;

    // Compute minimum coins required for all

    // values from 1 to V

    for (int i=1; i<=V; i++)
    {
        // Go through all coins smaller than i
        for (int j=0; j<m; j++)
          if (coins[j] <= i)
          {
              int sub_res = table[i-coins[j]];
              if (sub_res != INT_MAX && sub_res + 1 < table[i])
                 table[i] = sub_res + 1;
          }
   }
    return table[V];
}
// Driver program to test above function
int main()
{
    int coins[] =  {9, 6, 5, 1};
    int m = sizeof(coins)/sizeof(coins[0]);
    int V = 11;
    cout << "Minimum coins required is "
         << minCoins(coins, m, V);
```

```
     return 0;

 }
```

Output:

```
Minimum coins required is 2
```

Time complexity of the above solution is O(mV).