Greedy

Definitions

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.

Like dynamic-programming algorithms, greedy algorithms typically apply to optimization problems in which a set of choices must be made in order to arrive at an optimal solution.

The idea of a greedy algorithm is to make each choice in a locally optimal manner.

We hope that these choices will lead to a globally optimal solution.

It is not always easy to tell whether a greedy approach will be effective.

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.

A greedy method is used to solve optimization problems; problems that require either minimum or maximum resources.

Conditions of Greedy Coding

1. Optimal substructure
2. Optimal sub-problems

3. Greedy choice property

A locally optimal choice is globally optimal.

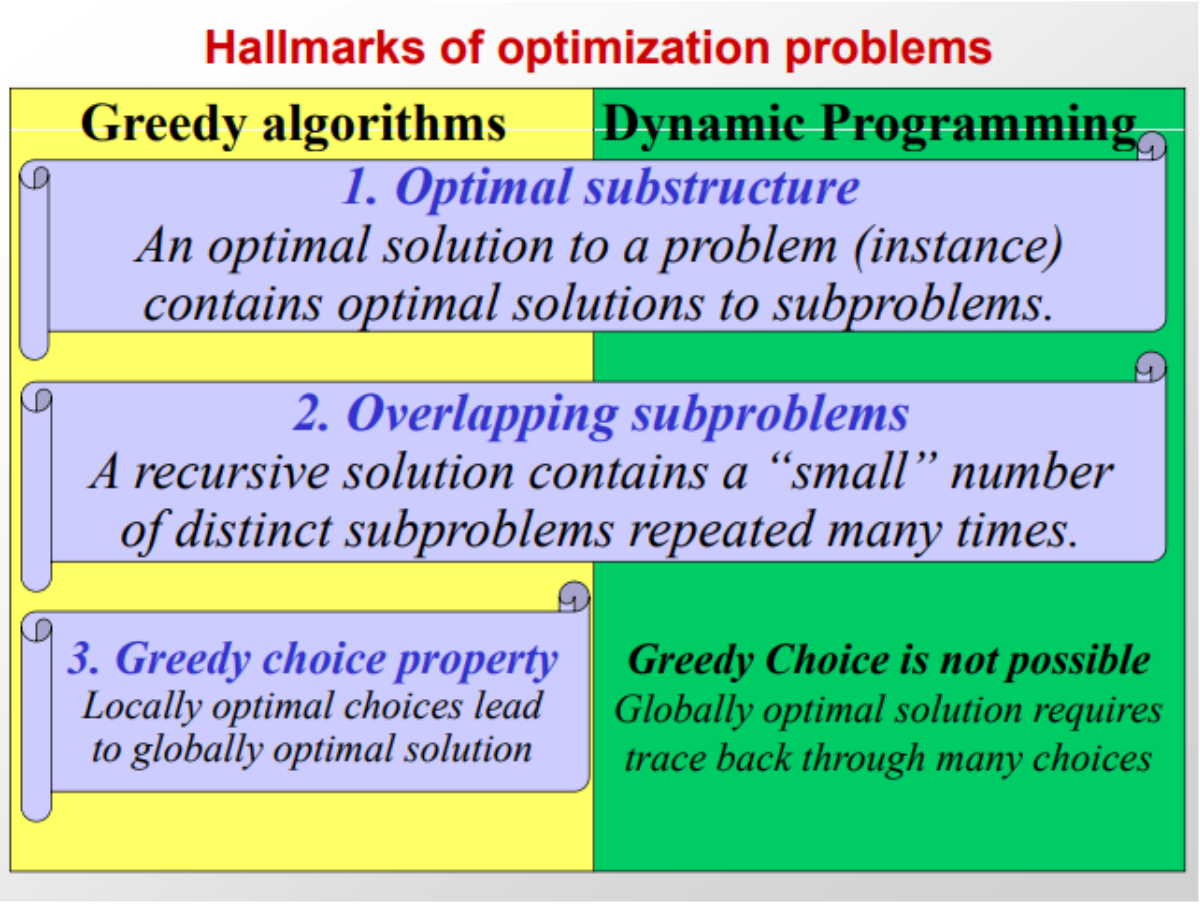
Unlike dynamic programming, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one.

A problem should be solved in stages. In each stages we will consider 1 input from the given problem and IF input is feasible (satisfying constraints), then we will include it in solution, thereby getting the optimal solution.

Greedy algorithms choose the best option to begin with and continues from there without going back (you don't look or record previous data)

```
Algorithm Greedy (a,n){
for( i=1 to n){
   x=select(a);
   if feasible(x)
       Solution+=x;
}
}
```

Difference Btw Dynamic and Greedy:



Huffman Codes

LINK: https://www.programiz.com/dsa/huffman-coding

- Huffman codes are a widely used and very effective technique for compressing data
- It is a Huffman technique.
- Huffman coding is a lossless data compression algorithm.
- savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.
- The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.
- The variable-length codes assigned to input characters are **Prefix Codes**, which means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character.
- This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit-stream.

PREFIX RULE: A code associated with a char should not be present in the **prefix** of any other code. Huffman ensures that this property is maintained.

It's used for reducing the size of the data or message, without losing any useful data. Generally useful on data with **FREQUENTLY OCCURING CHARACTERS.**

Sort in highest frequency by constructing a binary tree.

Step 1: Calculate the frequency of each character.

Step 2: Pick 2 lowest frequency nodes from the list, the sum of their frequency will be the frequency of parent nodes.

The best way to implement this is **Priority Queue**.

Repeated until we reach root 1 single node.

Step 3: Keep doing that until you reach the node storing the total number of char.

Every left edge of the tree is assigned the value 0 and every right edge is assigned 1.

Step 4: Read the codes from the root all the way down to the character's left node.

Selection Activity

Lab 10-problem 1

Example 1 : Consider the following 3 activities sorted by by finish time.

start[] = {10, 12, 20};

finish[] = {20, 25, 30};

A person can perform at most two activities. The maximum set of activities that can be executed is {0, 2} [These are indexes in start[] and finish[]]

Way of solving:

First: Sort the array according to the finish time.

Second: take i=0 and print start[i] and finish[i]

Third: for j=1 where j<tasks and j++

Fourth: Check if start[j]≥finish[j]

Fifth: if true print start[j] and finish[j]

Sixth: i==j and j++

Time complexity: O(n logn)

Code:

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.Scanner;
class selectionActivity{
   public static void main (String []args) {
     Scanner sc=new Scanner(System.in);
     int tasks=sc.nextInt();
     int[] s=new int[tasks];
     int[] f=new int[tasks];

     for(int i=0;i<tasks;i++)
       s[i]=sc.nextInt();

     for(int i=0;i<tasks;i++)
       f[i]=sc.nextInt();

     selectionActivity[] a=new selectionActivity[tasks];
     for(int i=0;i<tasks;i++)
        a[i]=new selectionActivity(s[i],f[i]);
     maxa(a,tasks);
```

```
     sc.close();
}

private int s;
private int f;

public selectionActivity(int start, int finish){
    this.s=start;
    this.f=finish;
  }

static void compare(selectionActivity a[], int n){
    Arrays.sort(a, new Comparator<selectionActivity>(){
      public int compare(selectionActivity m, selectionActivity n) {
      return m.f-n.f;
      }
      }
    );
  }

static void maxa(selectionActivity a[], int n){
    compare(a,n);
    int i=0;
    System.out.print(a[i].s+"-"+ a[i].f);
    System.out.println();
    for(int j=1;j<n;j++) {
      if(a[j].s>=a[i].f) {
        System.out.print(a[j].s+ "-"+ a[j].f);
        i=j;
        }
      System.out.println();
      }
    }
}
```

Frog Jumping